(2)

THE ECONOMICS OF SOFTWARE REUSE

MDA 972-92-J-1018

SPC-92119-CMC

VERSION 01.00.00

SEPTEMBER 1991

THE ECONOMICS OF SOFTWARE REUSE

SPC-92119-CMC

VERSION 01.00.00

SEPTEMBER 1991

Robert D. Cruickshank
John E. Gaffney, Jr.

Reprinted for the
VIRGINIA CENTER OF EXCELLENCE
FOR SOFTWARE REUSE AND TECHNOLOGY TRANSFER

February 1993

SOFTWARE PRODUCTIVITY CONSORTIUM, INC.
SPC Building
2214 Rock Hill Road
Herndon, Virginia 22070

CONTENTS

FIGURES

TABLES

PREFACE

This report presents a software reuse economics model which can be used to demonstrate the economics benefits of various types of reuse in software development. Return on investment in a domain is presented. The costs of incremental domain engineering, i.e., for investment in a domain, are calculated for various investment schemes, and the cost of money is considered. The cost impacts of reusing software requirements or design in addition to code are discussed. The effect of reuse on software quality is described. One can use the model to understand some of the economics effects of software reengineering.

There are three principal aspects of the reuse process from which significant economics benefits are derived:

- Reuse of artifacts (requirements, design, code)
- Incremental funding of the investment in domain engineering
- Systematic reuse (e.g., Synthesis)

The reuse economics model is based on earlier Consortium work (Gaffney and Durek 1988; Gaffney 1989; Cruickshank and Gaffney 1990) on the economics of software reuse. It builds and expands on that work to develop a theory that includes such features as various types of reuse including systematic reuse, Synthesis (Campbell 1990; Campbell, Faulk, and Weiss 1990), efficiency of reuse library structures, the concept of the number of break-even systems, return on investment, incremental funding of investment, and reuse of various software objects. All of these aspects of reuse can be viewed in terms of their effects on quality, cost (productivity), and development schedule. Thus, the reuse economics model is useful not only as a means to demonstrate benefits, but as a tool to aid the financial analyst, manager, or software engineer to better understand reuse and software domain analysis. This model will aid in decision making in all of these areas.

The reuse economics model will continue to evolve. As knowledge of systematic reuse expands and as experience with reuse applications increases, the model will grow and change to reflect that increased knowledge. The model presented in this report will be expanded to provide a basis for obtaining additional insight into the reuse process and to serve as an aid to member companies in expanding the degree of reuse that they practice. One significant extension of the treatment of the economics of reuse will relate aspects of application domain characterization to the costs of the activities that constitute systematic reuse.

The estimates of reuse costs, productivity, return on investment, break-even number of systems, and incremental funding schemes presented in this report are just that—estimates. The rigorous mathematical modeling presented here makes those estimates more precise, but mathematics cannot make estimates into certainties. When software reuse is applied in real-life software development efforts and when the

data resulting from those situations is fed back to the economics model, then the estimates will be more certain. At that point, the reuse economics model will clearly demonstrate its utility and its power.

The Consortium will deliver a computer spreadsheet model in 1991 which will implement some of the models described in this report.

ACKNOWLEDGEMENTS

The authors wish to thank Dr. Bill Frakes, Dr. Clem McGowan, and Tim Powell for their review of a draft of this report and the valuable commentary they provided.

This page intentionally left blank.

1. INTRODUCTION

1.1 BACKGROUND

Much attention has been paid to software reuse in recent years because it is recognized as a key means for obtaining higher productivity in the development of new software systems. Also, software reuse has provided the technical benefit of reduced error content and thus higher quality. The primary economics benefit of software reuse is cost reduction. Reuse of an existent software object generally costs much less than creating a new one.

An earlier Consortium technical report (Cruickshank and Gaffney 1990) presented an economics model of the Consortium's Synthesis system for systematic software reuse. This report extends this work.

The reuse economics model presented here should be regarded as a tool to aid in the exploration of the economics benefits of software reuse but not as an algorithm that covers all possible cases of reuse. The framework provided will aid the analyst and the project manager in making economics decisions about software reuse. The model covers various topics, including the effect of various strategies of investing in the creation of reusable software objects (RSOs), the cost effects of reusing requirements or design in addition to the costs of reusing code, and the effects of reuse on software quality.

1.2 AUDIENCE

This report addresses those interested in having a quantitative view of the benefits of software reuse and those who are responsible for estimating the impact of reuse on software development in their organizations. Those who desire a top-level view should read Sections 1 (Introduction) and 6 (Summary). These sections explain the purpose and approach to the economics of software reuse and outline the principal results. They do not require much mathematical knowledge to be understood. Readers who are interested in the details of the economics model should read the body of the report which requires an elementary knowledge of basic algebra, statistics, and economics.

The report is directed principally to the business area and financial managers and to cost and measurement analysts. The business area and financial managers must have a general understanding of the subject matter of the report so they can authorize studies of the economics impact of reuse on software development projects, direct such studies, and evaluate the results of such studies and make decisions based on those results. The cost and measurement analysts must have a detailed understanding of the techniques presented in the report so they can implement studies of the economics impact of reuse on software development through data collection, analysis, and economics modeling.

In addition to the above groups there is a wider audience for the report. The systems and software development community also has an interest in the economics impact of software reuse since the

economics and technical success of their projects is affected by reuse. Senior systems and software managers do not need to be concerned with the details of the techniques presented, however they do need a general understanding of how software reuse is implemented in their development environment and of the benefits derived from reuse. Operational software managers and lead software engineers are concerned with the details of the models and techniques presented because they provide the data for economics analyses, and they evaluate the impact of the level of reuse, both present and planned, on their development process. The report sections of primary interest to each of these groups are shown below.

Functional Responsibility	Applicable Section					
	1	2	3	4	5	6
Senior Systems and Software Managers	X					X
Business Area and Financial Managers	X	X	X			X
Software Project Managers	X	X			X	X
Lead Software Engineers	X	X	X	X		X
Cost and Measurement Analysts	X	X	X	X	X	X

Functional responsibilities are defined as follows:

Responsibility	Definition
Senior Systems and Software Managers	Division or Corporate Vice-President, or equivalent, responsible for authorizing measurement program across all software projects. Responsibility includes authorizing both direct costs and indirect (overhead) expense justifying economics basis of the project. Also includes high-level project managers with hardware and software responsibilities who want a general understanding of the economics and the benefits of software reuse.
Business Area and Financial Managers	Responsible for budgeting, financing, and tracking the cost status of software products and for authorizing studies of economics impact of alternative processes, environments, methods of financing, and return on investment,.
Software Project Manager	Person responsible for managing a software-based project. A project manager uses the economics of software reuse to estimate and control the economics aspects of software reuse in the project and to evaluate the economics advantages of various reuse schemes.
Lead Software Engineer	Technical supervisor, responsible for development or support of a software-based system. Supervises use of prescribed methods to perform technical activities.
Cost and Measurement Analysts	Technical staff members responsible for collecting project cost, size, and schedule status data, and for analyzing and projecting technical and economics project performance.

1.3 MODEL OBJECTIVES

The principal objective of the economics model of software reuse is to provide a means for understanding software reuse methodology and its economics impact. The economics model also provides a means to demonstrate, through a mathematical representation of various reuse schemes, the benefits derived from software reuse.

The report presents a set of techniques that can be applied to software development projects, actual or proposed, that enable the analyst to determine the economics impact of reuse on a specific software development process. To aid in the application of these techniques, the report gives examples.

1.4 BENEFITS OF THE REUSE ECONOMICS REPORT

The principal benefit of the report is an improved understanding of reuse processes by the audience, including systematic reuse. The economics analysis of reuse describes the generally beneficial effects of reuse on development costs, on the return on investment, on the costs of investment strategies of incremental domain engineering, on the cost effects of reusing requirements or design, and on software quality.

The two primary benefits of the report are:

A. To provide business area and financial managers with an understanding of:

- The effect of systematic reuse on development costs.
- The nature of return on investment in domain engineering.
- The cost effect of borrowing to finance domain engineering.
- Alternative funding approaches for domain engineering.

B. To provide software project managers and cost and measurement analysts a set of techniques which can be applied to projects, actual and proposed, to discover and explore:

- Systematic reuse.
- The cost effects of software reuse in systematic reuse applications.
- How reuse costs compare with those of current development approaches.
- The cost effect of incremental domain engineering.
- The cost effects of reusing software objects in addition to code in systematic reuse and reengineering.
- The effect of reuse on software product quality.

1.5 REUSE OVERVIEW

Software reuse can occur at many levels, ranging from the reuse of small granules of function (small software objects) within an application system to the reuse of large granules (large software objects) of software function across many application systems. For example, in an antiballistic missile system, the filtering routine in the signal processing function is a small granule while the location and tracking function is a large granule. The reuse methodology covers a wide range, from the 'ad hoc' level of reuse of code to the systematic reuse of software based on an application domain.

Reuse within an application system often takes place as the multiple use of a unit (or granule as above), such as a routine to implement a sine function or a finite impulse response filter, in a number of the major functions of that system. This type of reuse or multiple use of a software object has been common since FORTRAN began to be used. Multiple use within a system is facilitated in Ada through the use of the *with* and *include* constructs.

The reuse economics model presented here focuses on the systematic reuse of RSOs having a relatively large amount of functionality. These RSOs are not typically used more than once in a given application system. Systematic reuse is concerned with defining and establishing a domain of software systems, i.e., a family of software systems having similar descriptions (Parnas 1976). Such a family is a set of systems with similar requirements that can be (or are) satisfied by a common architecture and represent a set of closely related design choices at the detailed level. A domain is a coherent business area and the application area corresponding to the family of systems. A domain model characterizes an application family.

The benefits of establishing such a software domain are that software engineering and domain expertise are captured in a manageable form, and this knowledge can be used to produce families of similar application systems. As shown by Parnas, large (functional scale) RSO reuse can be sequential (from one application system to another) or parallel. In the latter case, a common set of RSOs may be used by several application systems which could be developed in parallel or sequentially. This type of reuse might actually be better termed multiple use. The Synthesis development methodology (Campbell 1990; Campbell, Faulk, and Weiss 1990) is concerned with this type of reuse.

The principal economics benefits of software reuse are:

- Lower development costs.
- Higher software product quality due to increased opportunity for error discovery.
- Reduced development schedule due to a reduced amount of development work.
- Reduced life-cycle costs due to reduced maintenance costs.

Systematic reuse views software maintenance as a series of redevelopments (i.e., incremental refinements) of application systems.

1.6 ACTIVITY-BASED MODELS

The economics model presented in this paper is an activity-based model (Cruickshank and Lesser 1982; Gaffney 1983; Software Productivity Consortium 1991). It considers the systematic reuse process in terms of the activities that comprise it. The model takes the view of industrial engineering when analyzing the activities of an industrial process, i.e., the individual activities are analyzed for their unique characteristics and then the set of activities are analyzed for their sequence and overall characteristics. The economics model is constructed in much the same way.

The total cost (TC) of a new application system is calculated as the sum of the costs of its new and reused software components added across the n activities that compose the development process. Algebraically, this concept can be represented by:

$$TC = \sum_{i=1}^{n} (LM/KSLOC)_{i,new} * KSLOC_{new} + \sum_{i=1}^{n} (LM/KSLOC)_{i,reused} * KSLOC_{reused}$$

The economics model is quantitatively expressed in labor rates measured in labor months per thousand source statements (LM/KSLOC). An alternative expression would be in hours per SLOC.

The unit cost of each activity in the systematic reuse process is expressed in LM/KSLOC. Labor rates are used in the economics model because they are additive, and the model is a linearly additive model. Labor rates and unit costs are conceptually the same and both are measured in LM/KSLOC. If function points were used as the unit of software size, the labor rates would be in labor months per function point.

The alternative to labor rates is productivities, which are usually expressed in SLOC/LM. Productivities are not additive and thus are not suitable for the model; however, the results of spreadsheet simulations are presented as productivities to allow for quick comparison with other models and projects. The conversion between the two forms is:

$$\text{LM/KSLOC} = 1000/(\text{SLOC/LM})$$

1.7 ASSUMPTIONS

The principal assumptions implicit in the reuse economics model are:

- Costs may be measured in labor months (LM).
- The true development cost for a new application system consists of the investment costs in domain engineering (apportioned over the expected number of application systems to which it applies) plus the cost of application engineering specific to the given application system.

  It is important to note that a development organization, under some circumstances, may not take the cost of domain engineering into account (as discussed in Section 3.2.4). One such situation is when a government agency provides the results of domain engineering to a contractor developing a new application system as government-furnished information.
- A new application software system is composed of two code categories, new and reused.
- A variety of software objects, including requirements, design, code, test plans, and test steps, may be reusable.
- The cost (in LM) of software development activities can be calculated as the product of a labor rate (LM divided by the size of the software product) and the size (in thousands of source statements) of the software product. Algebraically, this concept is represented by:

$$\text{LM} = (\text{LM/KSLOC})(\text{KSLOC})$$

This page intentionally left blank.

2. THE ECONOMICS MODEL

2.1 SYSTEMATIC REUSE

The reuse economics model presented here focuses on the systematic reuse of large-scale functional objects. Following the terminology developed by the Synthesis efforts, systematic reuse in the reuse economics model is viewed as consisting of two principal activities, domain engineering and application engineering. Domain engineering is the set of activities that are involved in creating RSOs that can be employed in a number of specific software systems or application systems. Application engineering is the set of activities that are involved in creating a specific application system.

Domain engineering is regarded in the economics analysis methodology presented here as covering the capital investment required to create a set of RSOs. Thus, domain engineering includes the capital investment activities necessary to produce a family of application systems. In domain engineering, the requirements for the family of software systems are identified, and the reusable structure to implement the family members is developed.

Capital investment here means the initial investment in terms of effort to create the means to produce application systems before those application systems are actually produced. This investment may be made all at once for the entire domain investment, or it may be made incrementally over the life of the domain, i.e., as long as the domain is used to produce application systems. The effort spent in domain engineering is a capital investment in creating the domain, including the domain definition and models, the application modeling language, and the reuse library. The term capital investment here does not imply any specific contractual arrangement.

2.1.1 Domain Engineering

Domain engineering is the capital investment process for creating the RSOs for a family of similar systems. It may be done up-front, all at once, or incrementally, over part or all of the time period. The family of application systems, which include some of the RSOs created by the domain engineering processes, is created in this time period. Domain engineering includes all of the activities associated with identifying a target family of application systems, describing the variation among these systems, constructing an adaptable design, and defining the methods for translating requirements into application systems composed of reusable components.

Domain engineering may not occur in some modes of reuse. One such mode is the ad hoc RSOs that were created in another application system. Such RSOs can include requirements and/or design and/or test plans as well as code. Alternatively, although domain engineering may occur, its cost may not be a consideration to the application system developer because it is borne by someone else. An example of this situation is when a government agency provides the RSOs produced by one contractor to another contractor tasked with developing an application system.

As shown subsequently, the costs of domain engineering may be amortized in different ways. The simplest way is to spread them across all of the planned application systems. Other methods include spreading some of them over one subset of the application systems or another part over another subset. The latter scheme is more realistic. It is often difficult if not impossible to envision all of the possible variations that might occur across a set of application systems. Also, it may be difficult to obtain sufficient funding to cover all of the domain engineering required for the family of application systems.

2.1.2 Application Engineering

Application engineering is the process of composing a particular software system which is a member of the family of systems defined in the domain engineering process. Application engineering consists of composing the specific application system with RSOs and any new software needed, reengineering existent software required, and testing the system.Thus, application engineering is a process for producing quality software from reusable components. The application systems are generated from reusable components to implement all of the associated requirements definitions.

Application engineering may be summarized as:

- Transforming the customer's input into a requirements specification for the specific application system to be developed.
- Generating new software objects specific to this application system, some of which may be reusable in other application systems and which may be entered into the reuse library.
- Composing the application system by integrating the new software objects and the reusable software objects obtained from the reuse library. The reuse economics model presented here considers any modified code to be in the new code category.

2.2 THE BASIC ECONOMICS MODEL OF SOFTWARE REUSE

This section presents the basic model of software reuse. The first version of the model, the basic model, assumes up-front domain engineering. The second version of the model covers incremental domain engineering.

2.2.1 Reuse Economics Model With Up-Front Domain Engineering

The reuse economics model is designed to reflect the total costs of applying a reuse scheme. The model treats the cost of an application system as the cost of the capital investment in domain engineering apportioned over the expected N application systems plus the cost of application engineering (the cost of creating that particular system). Thus, the cost of an application system, C_S, equals the prorated cost of domain engineering plus the cost of application engineering. Further, the cost of application engineering is the cost of the new code plus the cost of the reused code in the new application system, and R is the proportion of code that is reused code. Then:

$$C_S = C_{DP} + C_A$$

$$C_S = C_D/N + C_N + C_R$$

where:

$$C_{DP} = C_D/N \text{ and } C_A = C_N + C_R$$

- C_S = The total cost of an application system.
- C_D = The total cost of domain engineering.
- C_{DP} = The pro rata share of domain engineering borne some by each of the N application systems.
- C_A = The cost of an application system.
- C_N = The cost of the new code in the application system.
- C_R = The cost of the reused code in the application system.

Each of the costs, C_D, C_N, and C_R, is the product of a unit cost (LM/KSLOC) and an amount of code (KSLOC). Note that all costs are in LM.

Then:

$$C_D = C_{DE} \bullet S_T$$
$$C_N = C_{VN} \bullet S_N$$
$$C_R = C_{VR} \bullet S_R$$

Therefore the basic reuse cost equation is:

$$C_S = C_{US}\, S_S = C_{DE}\, S_T / N + C_{VN}\, S_N + C_{VR}\, S_R$$

where:

- C_{US} = Unit cost of the application system.
- C_{DE} = Unit cost of domain engineering.
- C_{VN} = Unit cost of new code developed for this application system.
- C_{VR} = Unit cost of reusing code from the reuse library in this application system. It represents the unit cost of reused code in the case where the library components can be instantiated directly into the application system with no modification.
- S_T = Expected value of the unduplicated size of the reuse library, i.e., the available, reusable functionality (software objects measured in source statements) in the library.
- S_N = Amount of new code in source statements developed for this application system.
- S_R = Amount of reused code (from the reuse library) incorporated into this application system in source statements.
- S_S = Total size of the application system in source statements.

Code sizes S_N, S_R, S_S, and S_T are denominated in source statements, either physical or logical (Gaffney and Cruickshank 1991a; Gaffney and Cruickshank 1991b). These code sizes could be denominated in function points (Albrecht and Gaffney 1983) or their variations, such as feature points. The important thing is that consistent units of code size are used.

Let $S_N/S_S = 1 - R$ and $S_R/S_S = R$, where R is the proportion of reuse.

Dividing through by S_S and rewriting:

$$C_{US} = \frac{C_{DE}S_T}{NS_S} + C_{VN}(1-R) + C_{VR}R$$

Now let $S_T/S_S = K$, the library relative capacity. Thus:

$$C_{US} = \frac{C_{DE}}{N} \cdot K + C_{VN} - (C_{VN} - C_{VR}) \cdot R$$

This is the basic reuse unit cost equation. It presumes a single reuse of S_R units (SLOC, KSLOC, function points) in each of the N application systems, on the average. Thus, this equation is most applicable to systematic reuse of units of code having a relatively large amount of functionality.

Some of the software developed for a given application system, of amount S_N, might be deemed reusable on other application systems. Such software may be treated as resulting from a portion of an incremental domain engineering investment.

Although not treated further here, the unit cost parameters (C_{VN}, C_{VR}, and C_{DE}) can be considered to be time-variant. For example, they can represent the effects of technology change (methodology and tools) over time. These parameters are considered to be time-invariant here.

2.2.2 Library Efficiency

This section discusses some aspects of the structure of a reuse library from an economics point of view.

A reuse library may be constructed so that there are a number of alternative or duplicate units of code (or RSOs) to cover the expected variation of a unit of function. Alternatively, there may be just one unit of code (or RSO) per function, but with the expected variation to be covered by the (application engineer's selection of the) values of one or more parameters to cover that variation.

S_T is the "unduplicated" size of the library or its capacity. There may well be alternate or duplicate implementation functionality in the reuse library (source codes, as just stated), but that alternate or duplicate functionality does not add to the size of S_T. The case of alternative implementation of source code or all of the functionality of size S_T is covered in the cost model by an appropriate selection of the value of the unit cost parameter, C_{DE}.

The factor K ($= S_T / S_S$), the library relative capacity, represents the average proportion (over the N application systems) of an application system in the family of systems that the library covers. Thus, if S_S represents the average application system size in the domain of interest, K is the upper bound for R, or $R \leq K \leq 1$.

The efficiency of the library infrastructure, E, is the ratio of the amount of reused code in the application system to the available reusable code.

$$E = \frac{R}{K} = \frac{S_R / S_S}{S_T / S_S} = \frac{S_R}{S_T}$$

where $0 \leq E \leq 1$.

The factor E indicates the extent to which the developer of a new application system has been able to make use of the library of reusable components in the new system. For example, the reuse library may contain a Kalman filtering program and a navigation program that contains a Kalman filtering routine. If the Navigation program is selected (perhaps because it contains a Kalman filtering routine) for use in an application system, then the efficiency of the library for that specific application system is less than 1.0 because the alternate (or duplicate) Kalman filtering program was not used.

E is a measure of the systematic reuse application process efficiency. Normally E is 1.0 or slightly less th. ⁊ 1.0, since application engineers on average are expected to reuse as much code as possible when composing an application system.

If K is assumed to be equal to R, or $S_R = S_T$ (which means E = 1), then the basic reuse unit cost equation can be rewritten as:

$$C_{US} = \frac{C_{DE}}{N} * R + C_{VN} - (C_{VN} - C_{VR}) * R$$

Consolidating terms obtains:

$$C_{US} = C_{VN} - \left(C_{VN} - C_{VR} - \frac{C_{DE}}{N} \right) R$$

This equation is the basic reuse unit cost equation.

2.3 SOME EXAMPLE APPLICATIONS OF THE MODEL

This section provides three example applications of the basic reuse unit cost equation. The three examples are an Ada aerospace system, a real-time command and control (RTCC) application, and a management information system (MIS) application. These applications have the values C_{DE}, C_{VN}, and C_{VR} given in LM/KSLOC appropriate to a specific instance of domain and application engineering. The labor rates for C_{VN} and C_{VR} are derived from actual RTCC, MIS, and Ada development experience. The labor rates for C_{DE} are based on analysis of the functions included in domain engineering for the RTCC and MIS applications. In the case of the Ada aerospace application, a value of 1.5 for the ratio of C_{DE} to C_{VN} is assumed. The RTCC labor rates (unit costs) are derived from experience based on a DOD-STD-2167A model translated to a systematic reuse model. The MIS labor rates (unit costs) are based on experience with SPECTRUM* and with function points translated to the systematic reuse model derived above.

Table 1 shows the unit costs (in LM/KSLOC) of the three cost configurations.

Table 1. Cost Parameter Applications

Cost Parameters	Application (LM/KSLOC)		
	RTCC	MIS	Ada Aerospace
C_{DE}	5.305	2.122	15.000
C_{VN}	2.072	1.012	10.000
C_{VR}	0.514	0.271	1.000

* Trademark of Software Architecture and Engineering, Inc.

Thus, the three parametric configurations of the systematic reuse unit cost equations are:

$$\text{RTCC:} \quad C_{US} = \frac{5.305}{N} * K + 2.072 - 1.558 * R$$

$$\text{MIS:} \quad C_{US} = \frac{2.122}{N} * K + 1.012 - 0.741 * R$$

$$\text{Ada aerospace:} \quad C_{US} = \frac{15.000}{N} * K + 10.000 - 9.000 * R$$

Table 2, which shows the productivities resulting from these three configurations, illustrates the cost and productivity benefits gained from systematic reuse. Available data shows industry productivities for new software development (design through integration test) to be in the range of 80 to 180 SLOC/LM (12.500 to 5.556 LM/KSLOC). The reuse productivities in Table 2 show a considerable improvement over these performances.

Also note that, where the value of R increases in Table 2, the productivity actually decreases for certain values of N. This result is contrary to intuition, which would expect increasing productivity to accompany increasing values of R. However, where the number of expected application systems is less than the number of break-even systems, decreasing productivity accompanies an increasing proportion of reuse. This phenomenon is discussed in Section 3 where the concept of break-even systems is introduced.

Table 2. Economics Model Application Productivities

(E=1.0)		Application (SLOC/LM)		
N	R	RTCC	MIS	Ada Aerospace
2	0.7	352	809	112
2	0.9	327	769	116
3	0.7	451	1,011	139
3	0.9	442	1,018	156
4	0.7	524	1,156	158
4	0.9	537	1,215	190
5	0.7	580	1,265	172
5	0.9	615	1,375	217
10	0.7	739	1,557	211
10	0.9	872	1,864	308
15	0.7	814	1,687	227
15	0.9	1,012	2,115	357

2.4 SOME RECENT REUSE EXPERIENCE

This section provides some data on recent reuse experience. Because no formal domain engineering was done in the composition of these systems, the value for C_{DE} was set at zero. The systems were done in sequence, with software objects being reused (and modified in some cases) from a prior system in the creation of a new software application system.

2.4.1 Management Information Systems

Allan Albrecht (Albrecht 1989) provided some worldwide reuse experience from IBM in the development of MIS applications during the period of 1984 to 1988. The data is in the form of function point and productivity measurements on software created for internal IBM applications such as billing and ordering. The applications were written in PL/1. One function point (Albrecht and Gaffney 1983) is equivalent to about 80 lines of PL/1 or 106 lines of COBOL. The 1988 reuse data analyzed here was determined from about 0.5M function points from more than 50 development sites, worldwide.

Figure 1 presents this function point data on overall product productivity, new code productivity, and average percent reuse. Although the productivity data and the percent reuse data are measured on different scales, the range of the three sets of data could all share a common vertical axis. The overall product productivity and the percent code reuse figures are for the years 1984 to 1988. The new code productivity figures are for 1986 to 1988; data for the 1984 to 1985 period was not available. Note that overall productivity is equal to total function points in the software system divided by total LM, while new code productivity is equal to function points of new code per LM for new function points. Table 3 shows the data to which the histograms correspond.

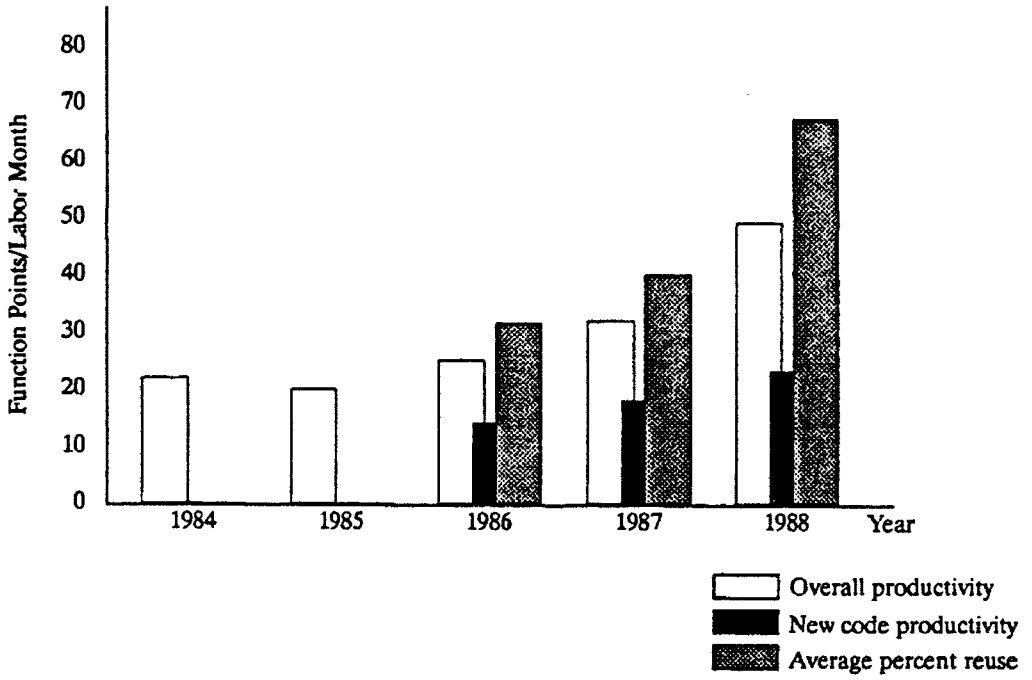


Figure 1. Worldwide Productivity and Reuse Experience

Table 3. Productivity and Reuse Experience

Year	Overall Productivity (P) Function Points/LM	New Code Productivity (N) Function Points/LM	Average Percent Code Reuse (R)
1984	22	—	—
1985	20	—	—
1986	25	14	31.5
1987	32	18	40.0
1988	49	23	67.2

Table 4 shows the partial correlations for the years 1986 to 1988 among the three variables and the corresponding figures for $100r^2$, the percentage variation of one variable "explained" by its relationship to the other and corrected for the third. Partial correlations indicate the correlations between two variables while holding the third constant, i.e., correcting for the third.

Table 4. Partial Correlations Among Variables, 1986–1988

Variables		Correlation r	$100r^2$
Correlated	Held Constant		
P,R	N	0.9982	98.36
P,N	R	0.9854	97.10
R,N	P	-0.9736	94.79

The strong partial correlations indicate that both the new code productivity (N) and the percent code reuse (R) had a strong influence on the increase in overall productivity (P) in the years 1986 to 1988. Table 5 shows the percent increase in each variable. There was an increasing level of P over the period shown both from the reuse of code and from the new code. This was partially based on the reuse of existing requirements or design.

Table 5. Percent Increase of Variables, 1986–1988

Variable	Percent Increase
P	96
N	64
R	113

Figure 2 presents a plot of unit cost, C_{US}, in LM per function point multiplied by 100, versus the proportion of code reuse for the software development sites reporting in 1988. The data was grouped into six ranges of reuse plus the point (0.0, 5.41), as presented in Table 6.

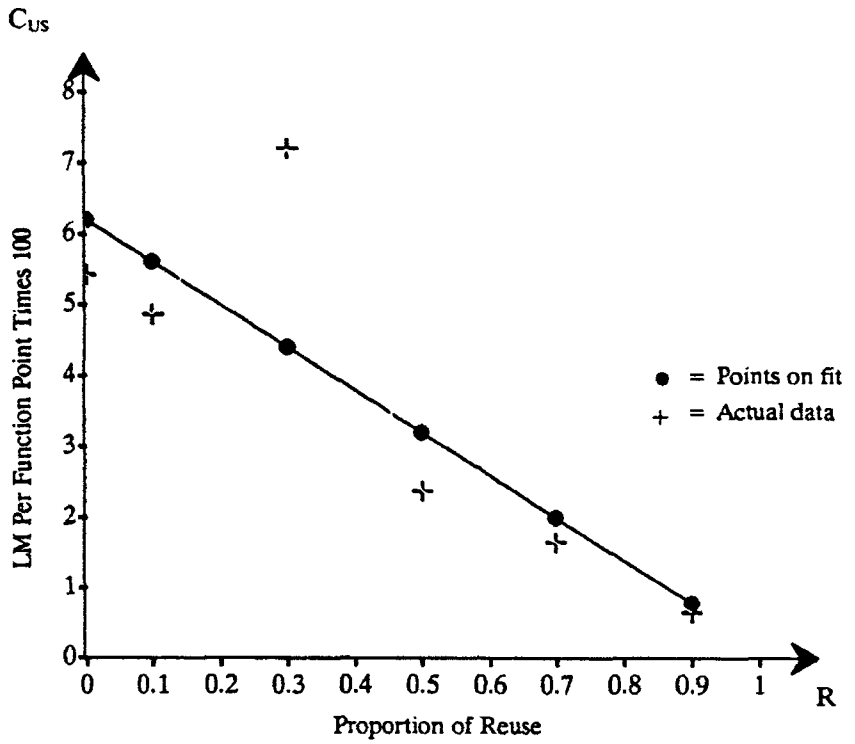


Figure 2. Cost Per Product Unit for 1988

Table 6. 1988 Product Unit Costs Versus Proportion of Code Reuse

Proportion of Reuse, R	(LM/Function Point) Times 100
0.0	5.41
0.1	4.85
0.3	7.19
0.5	2.35
0.7	1.63
0.9	0.63

The product moment sample correlation of C_{US} and R was found to be -0.832 (significant at the 5 percent level), which means that 69.16 percent of the variation in C_{US}, the overall product unit cost, was "explained" by its relationship with R, the proportion of reuse. The regression equation was:

$$C_{US} = 6.188 - 6.027 * R$$

C_{VR} should not be estimated from the relationship:

$$C_{USi} = C_{VN} - (C_{VN} - C_{VR}) * R + \epsilon_i$$

i.e., using the relationship based on least squares regression as shown previously, because it provides estimates of C_{VN} and (C_{VN}-C_{VR}) and not of C_{VR}. Instead the statistical cost relationship:

$$C_{Ai} = C_{VN} * S_{Ni} + C_{VR} * S_{Ri} + \epsilon_i$$

based on the general linear hypothesis of full rank can be used to calculate values for C_{VN} and C_{VR}.

In order to get a more nearly complete picture of the costs involved in reuse, as stated earlier, the cost of reusable code creation and the cost of domain engineering must be determined (and presumably, amortized over the set of users).

2.4.2 Aerospace

Figure 3 shows total unit cost in LH/SLOC, C_{US}, plotted against the percent of code reuse, R, for eight technical software applications from the aerospace industry. The 0 percent data point is the average of five points, 0.6433 LH/SLOC. A straight line has been fitted using linear regression, and the fitted equation is:

$$C_{US} = 0.7850 - 0.009435 * R$$

The sample correlation of C_{US} and R is $r = -0.785$, which means that $100r^2 = 61.54$ percent of the variation in C_{US} is explained by its relationship with R. It is obvious from this data and from the fitted line that unit cost declines with an increasing proportion of reuse.

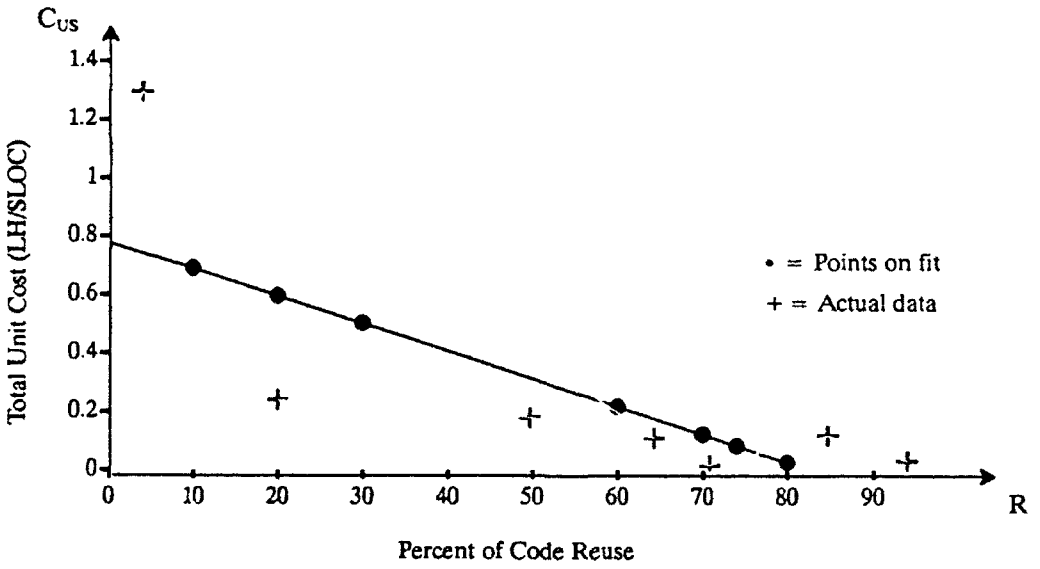


Figure 3. Unit Cost as a Linear Function of Percent Reuse

Figure 4 shows the same data as in Figure 3 with a quadratic form fitted. The equation is:

$$C_{US} = 0.920 - 0.0239 * R + 0.00016114 * R^2$$

Here the multiple correlation of C_{US} with R and R^2 is $r = -0.846$. Thus, the quadratic equation in R provides a better fit than shown in Figure 3 since only $100r^2 = 71.6$ percent of the variation in C_{US} is explained by its relationship to R and R^2 in that case. The goodness of this relationship suggests that, in some reuse regimes, the unit cost of software products decrease with increasing levels of reuse

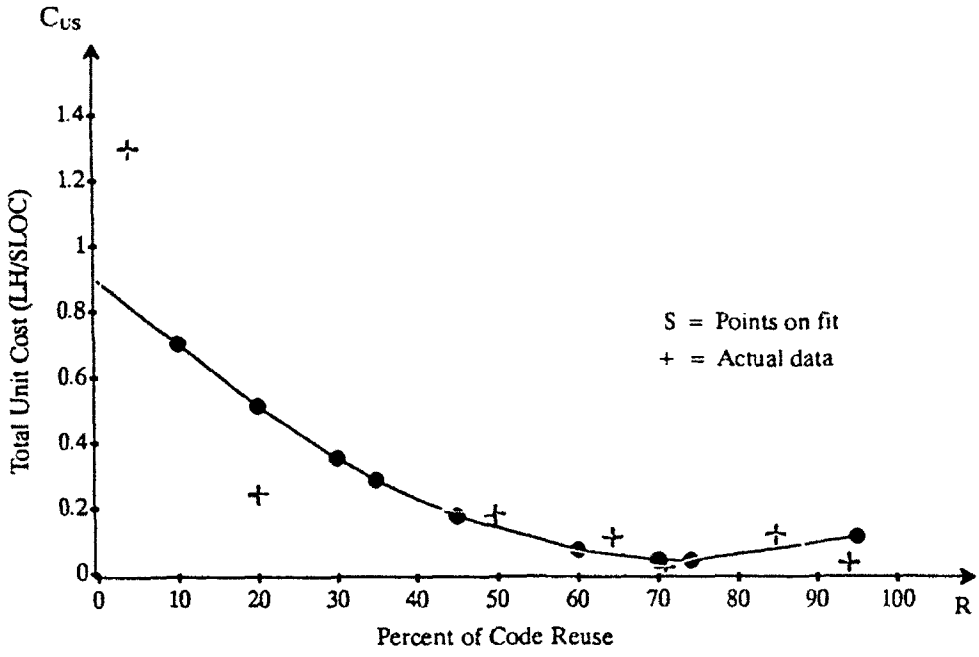


Figure 4. Unit Cost as a Quadratic Function of Percent Reuse

but then increase beyond a certain level of reuse. Perhaps the nature of the reuse process becomes less efficient beyond this point.

This page intentionally left blank.

3. RETURN ON INVESTMENT

3.1 UP-FRONT DOMAIN ENGINEERING

This section defines the break-even number of systems for the case in which all of the domain engineering is done "up front" as assumed in the basic reuse unit cost equation. The break-even number of systems, i.e., the number of application systems necessary to offset the cost of domain engineering, is discussed next.

3.1.1 Break-Even Number of Systems

Reuse pays off when the unit cost of an application system which includes reused software is less than or equal to the unit cost of an implementation in all new software. Therefore the break-even number of systems, N_0, is the value of N when $C_{US} = C_{VN}$. Using the basic systematic reuse unit cost equation developed in section 2:

$$C_{US} = C_{VN} - (C_{VN} - C_{VR})R + K\frac{C_{DE}}{N}$$

and dividing through by C_{VN} produces:

$$C = \frac{C_{US}}{C_{VN}} = 1 - \left(\frac{C_{VN} - C_{VR}}{C_{VN}}\right)R + \frac{C_{DE}}{C_{VN} \cdot N}K$$

Break-even costs occur when C = 1. Let the number of application systems required to break even be N_0. This would be:

$$0 = -\left(1 - \frac{C_{VR}}{C_{VN}}\right)R + \frac{C_{DE}}{C_{VN} \cdot N_0}K$$

or

$$\left(1 - \frac{C_{VR}}{C_{VN}}\right)E = \frac{C_{DE}}{C_{VN} \cdot N_0}$$

Then:

$$N_0 = \frac{C_{DE}}{(C_{VN} - C_{VR})^{*}E}$$

where $E = R/K$ is the efficiency in the use of the library content, $R = S_R/S_S$ and $K = S_T/S_S$, $S_T \leq S_S$, $R \leq K$, and $S_R \leq S_T$. Table 7 shows the break-even number of systems for values of E and for the quality levels in the three applications previously discussed.

Table 7. Break-Even Number of Systems

E=R/K	RTCC	MIS	Ada Aerospace
0.7	4.86	4.09	2.39
1.0	3.40	2.86	1.67

The situation of decreasing productivity with increasing R (illustrated in Table 2) occurred when the expected number of application systems, N, was less than the number of break-even systems for a particular application type. This phenomenon can be explained by a restatement of the basic unit cost system as:

$$C_{US} = C_{VN} + [-(C_{VN} - C_{VR}) + C_{DE}/EN]R$$

Since all unit costs are in LM/KSLOC, as R increases C_{US} will increase as long as:

$$C_{DE}/EN - (C_{VN} - C_{VR}) > 0$$

That is, the labor rate C_{US} in LM/KSLOC will increase with increasing R but productivity in SLOC/LM will decrease as long as the above inequality is true. Solving this inequality for N:

$$N < C_{DE}/[(C_{VN} - C_{VR})E] = N_0$$

As long as the expected number of application systems is less than the break-even number of systems, productivity will decrease with increasing R. N_0 depends only on the cost structure and not on R.

Since $E = S_R/S_T$, if $S_R = S_T$, the amount of reuse is the maximum possible and $E = 1$. Then, $K = R$ and the basic systematic reuse unit cost equation becomes:

$$C_{US} = C_{VN} - (C_{VN} - \frac{C_{DE}}{N} - C_{VR})^{*}R$$

In this case, the break-even number of systems, N_0, is found by setting $C_{US} = C_{VN}$, as before. For this to occur:

$$C_{VN} - \frac{C_{DE}}{N_0} - C_{VR} = 0$$

or

$$N_0 = \frac{C_{DE}}{C_{VN} - C_{VR}}$$

This is exactly the equation derived above but with E = 1.

Figure 5 shows the RTCC cost model application data from Table 2 plotted as productivity in SLOC/LM versus the number of application systems for proportions of reuse R = 0.7 and R = 0.9. This chart illustrates the phenomenon of higher reuse producing lower productivity when the number of application systems is below the break-even point. The MIS or Ada data from Table 2 could also be used to illustrate this phenomenon.

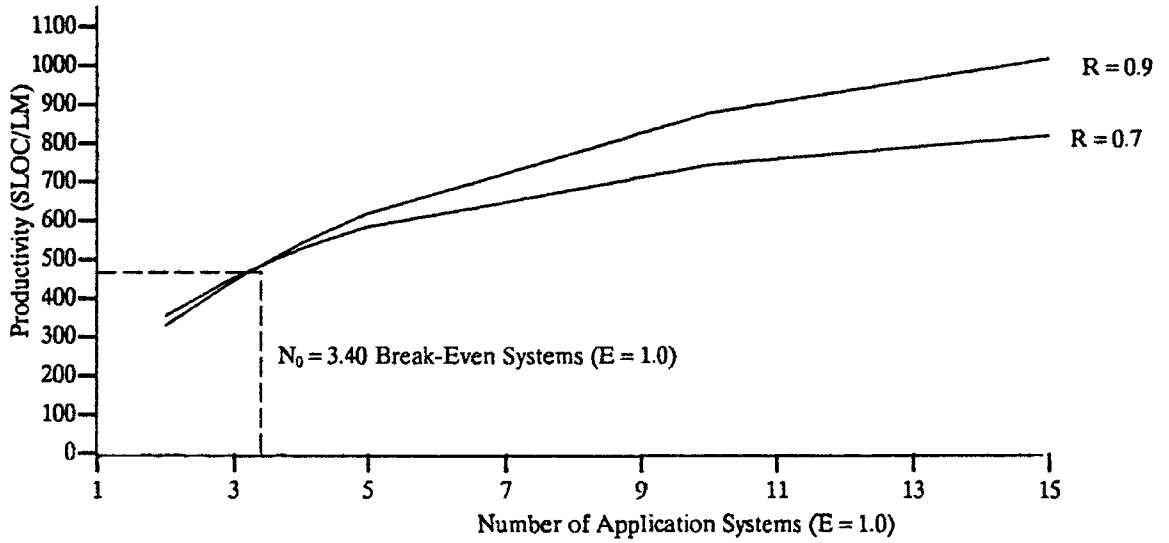


Figure 5. Number of Application Systems Versus Productivity at Two Levels of Reuse

Table 7 shows that when E = 1.0, the break-even number of systems for the RTCC cost model application example is 3.40. Let R = K = 0.9 since E = R/K = 1.0. Substituting these values into the basic unit cost equation for the RTCC application:

$$C_{US} = (5.305/3.40)(0.9) + 2.072 - 1.558(0.9) = 2.07 \text{ LM/KSLOC}$$

and 1,000/2.07 = 483 SLOC/LM. Therefore the 3.40 break-even number of systems corresponds to a productivity of 483 SLOC/LM, and whenever N is greater than 3.40, reuse pays off. Note that in the example case of Ada aerospace systems, the break-even number of systems (also when E = 1.0) is 1.67. That is, for N = 2 systems, reuse pays off.

3.1.2 Calculating Return on Investment

As was stated previously, the cost of domain engineering activities represents an investment in the systematic reuse process to achieve a high degree of reuse. The return on this investment is the difference in costs between the cost of N application systems in which there is no reuse and the cost of N application systems in which there is an average reuse of R. If the cost (in LM/KSLOC) of domain engineering is denoted as C_{DE}, the cost (in LM/KSLOC) of new software is denoted as C_{VN}, and the cost of reused software (in LM/KSLOC) is denoted as C_{VR}, then it can be shown that the percent return on investment (ROI) is:

$$ROI = \left[\frac{N * E * (C_{VN} - C_{VR})}{C_{DE}} - 1 \right] * 100$$

where N is the number of application systems and E is the efficiency factor discussed above.

The number of systems, N_0, at which the ROI is zero, may be termed the break-even number of systems. It is determined by setting:

$$\left[\frac{N_0 * E * (C_{VN} - C_{VR})}{C_{DE}} - 1 \right] = 1$$

Thus:

$$N_0 = \frac{C_{DE}}{(C_{VN} - C_{VR})E}$$

Therefore the expression for ROI may also be written as:

$$ROI = \left(\frac{N}{N_0} - 1 \right) 100$$

In the case of ROI, the emphasis is on determining when an investment in domain engineering pays off. This can be the case for relative productivity calculations as well. In addition, productivities relative to those of current industry practice may also be of interest, especially to those who wish to understand how systematic reuse compares with current practice.

Table 8 shows the comparison of return on investment for selected values of N. The negative values of percent return on investment are caused by the number of systems (N) being below the break-even number of systems.

Table 8. Percent Return on Investment (E = 1.0)

N	RTCC	MIS
2	-41.3	-30.2
3	-11.9	4.7
4	17.5	39.6
5	46.9	74.5
10	193.7	249.1
15	340.6	423.6

The equation for return on investment can be restated in terms of the following expression:

$$N = \left(\frac{ROI}{100} + 1\right)\left(\frac{C_{DE}}{C_{VN} - C_{VR}}\right)\left(\frac{1}{E}\right)$$

Figure 6 shows that, for the MIS and RTCC cost model applications, the higher the library efficiency, the greater the return on investment.

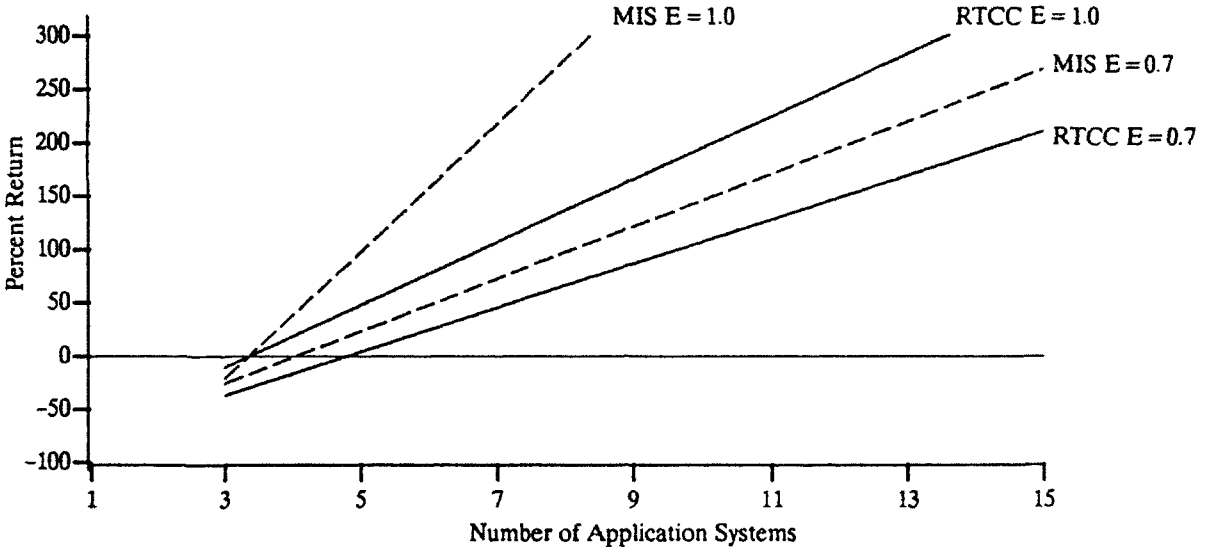


Figure 6. Number of Application Systems Versus Return on Investment

Suppose that a 20 percent return is the least return on investment that is acceptable, and suppose that a 50 percent return is considered the highest return that is possible. Let $C_{DE} = 5.305$, $C_{VN} = 2.072$, and $C_{VR} = 0.514$ as with the RTCC example discussed in Section 2. Then, using the above equation, N•E has the value 4.09 for the 20 percent return case and 5.11 for the 50 percent return case. The relationship between N and E then becomes as shown in Figure 7, and the 20 to 50 percent operating region is the area between the lines.

3.2 INCREMENTAL DOMAIN ENGINEERING

3.2.1 Break-Even Number of Systems

This section generalizes the basic reuse economics model presented earlier to cover the case in which the domain engineering is not done entirely at once, up front.

The basic reuse economics model implies that all of the domain engineering is complete before the first application system is produced. For certain domains and environments this may be the case, but domain engineering does not necessarily have to be done in this fashion. Domain engineering may be done incrementally, (i.e., piecewise), with some domain engineering being done in conjunction with more than one of the N application systems produced from the domain.

Consider the S_T KSLOC of unduplicated code in the reuse library that is to be instantiated into one or more of the N application systems to be produced from the domain. Suppose that S_{T1} KSLOC is developed in association with the development of system number 1, S_{T2} KSLOC is developed in

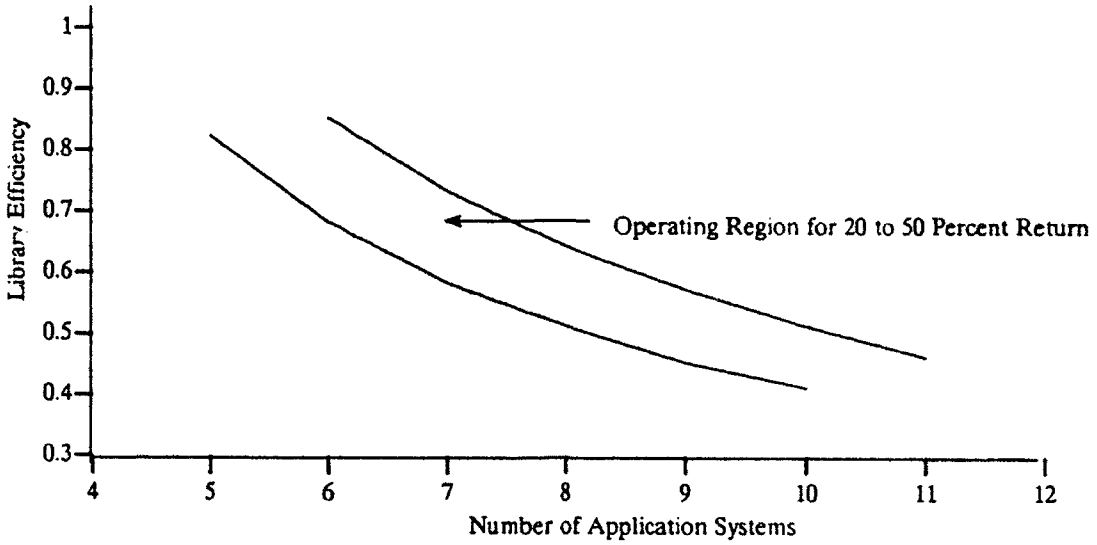


Figure 7. Number of Application Systems Versus Library Efficiency

association with the development of system number 2, and so on. In general S_{Ti} will be developed in association with the development of system number i. Thus $0 \leq S_{Ti} \leq S_T$ for $i = 1,...,N$ so that:

$$S_T = \sum_{i=1}^{N} S_{Ti}$$

Thus S_{T1} is amortized over N application systems, S_{T2} is amortized over N–1 systems, and in general S_{Ti} is amortized over N–(i–1) systems.

For the *i*th system out of N application systems, the unit cost is:

$$C_{USi} = \left(\frac{C_{DE}}{S_S}\right) \sum_{m=1}^{i} \left(\frac{S_{Tm}}{(N-(m-1))}\right) + C_{VN} - (C_{VN} - C_{VR}) \sum_{m=1}^{i} \left(\frac{S_{Tm}}{S_S}\right)$$

which reduces to:

$$C_{USi} = C_{DE} \sum_{m=1}^{i} \left(\frac{S_{Tm}}{(N-(m-1))}\right) + C_{VN}S_S - (C_{VN} - C_{VR}) \sum_{m=1}^{i} S_{Tm}$$

This is the basic unit cost equation with incremental domain engineering, for domain engineering occurring in more than one period of time. Note that E is assumed to be equal to 1.0. Consequently:

$$R_i = \sum_{m=1}^{i} \left(\frac{S_{Tm}}{S_S}\right)$$

which is the maximum amount of reuse possible for system i.

If $S_{T1} = S_T$ and if $S_{Ti} = 0$ for $i = 2,3,...,N$ (where i is not equal to j) then the above base unit cost equation for incremental domain engineering reduces to:

$$C_{USi} = \left(\frac{C_{DE}}{N}\right)\left(\frac{S_T}{S_S}\right) + C_{VN} - (C_{VN} - C_{VR})\left(\frac{S_T}{S_S}\right)$$

which is the same form as the basic cost equation with K = R, for the cost of up-front domain engineering.

For the *i*th system to break-even, its cost must be less than or equal to that for the case in which the entire system of size S_S would be made from entirely new code:

$$C_{USi} = C_{DE}\sum_{m=1}^{i}\left(\frac{S_{Tm}}{(N-(m-1))}\right) - (C_{VN} - C_{VR})\sum_{m=1}^{i} S_{Tm} \leq 0$$

If, as before, $S_{T1} = S_T$ and $S_{Ti} = 0$ for $i = 2,3,...,N$ (where i is not equal to j) the above equation reduces to:

$$\frac{C_{DE}}{(C_{VN} - C_{VR})} \leq N_0$$

which is of the same form as the break-even number of systems calculated from the basic cost equation with E = 1.

Now the expression for calculating the break-even number of systems for the more general case in which at least one of the $S_{Ti} > 0$, $i = 2,3,...,N$, is:

$$C_{DE}\sum_{i=1}^{N}\sum_{m=1}^{i}\left(\frac{S_{Tm}}{(N-(m-1))}\right) - (C_{VN} - C_{VR})\sum_{i=1}^{N}\sum_{m=1}^{i} S_{Tm} \leq 0$$

Now:

$$\sum_{i=1}^{N}\sum_{m=1}^{i}\frac{S_{Tm}}{(N-(m-1))} = N\left(\frac{S_{T1}}{N}\right) + (N-1)\left(\frac{S_{T2}}{(N-1)}\right) + \ldots + S_{TN} = S_{T1} + \ldots + S_{TN} = S_T$$

and:

$$\sum_{I=1}^{N}\sum_{m=1}^{i} S_{Tm} = NS_{T1} + (N-1)S_{T2} + (N-2)S_{T3} + \ldots + S_{TN}$$

Therefore, for the N application systems in the domain to break even overall:

$$\frac{C_{DE}}{(C_{VN} - C_{VR})} \leq \frac{(NS_{T1} + (N-1)S_{T2} + \ldots + S_{TN})}{S_T}$$

And the break-even number of systems, N_0, is found by solving the above equation for N.

Let:

$$S_{Ti} = a_i S_T, \qquad \sum_{i=1}^{N} a_i = 1$$

Then the right side of the above equation becomes:

$$Na_1 + (N-1)a_2 + \ldots + (N-(N-1))a_N = N - P$$

where:

$$P = \sum_{i=1}^{N} (i-1) * a_i = \sum_{i=1}^{N} i * a_i - 1$$

Thus the break-even number of systems, N_0, is given by:

$$N_0 = \frac{C_{DE}}{C_{VN} - C_{VR}} + P$$

where P is the incremental spending penalty, i.e., the extra number of application systems required to break even due to incremental domain engineering. It is clear that doing domain engineering incrementally has the effect of increasing the number of systems required to break even as compared with doing domain engineering all at once.

3.2.2 Calculating Return on Investment

Now four cases of incremental funding of domain engineering investment are presented. The value of P, the additional number of application systems for break even to occur, is calculated for each case with the formula provided above.

Case 1: $S_{T1} = S_T$

$$\frac{(NS_{T1})}{S_T} = N - 0, \quad \text{or} \quad P = 0$$

Case 2: $S_{T1} = S_{T2} = \frac{S_T}{2}$

$$\frac{(NS_{T1} + (N-1)S_{T2})}{S_T} = N - \frac{1}{2}, \quad \text{or} \quad P = 0.5$$

Case 3: $S_{T1} = S_{T2} = S_{T3} = S_{T4} = \frac{S_T}{4}$

$$\frac{(NS_{T1} + (N-1)S_{T2} + (N-2)S_{T3} + (N-3)S_{T4})}{S_T} = \frac{4N - (1+2+3)}{4} = N - \frac{3}{2}, \text{or} \quad P = 1.5$$

$$\text{Case 4: } S_{T1} = \left(\frac{5}{15}\right)S_T,\ S_{T2} = \left(\frac{4}{15}\right)S_T,\ S_{T3} = \left(\frac{3}{15}\right)S_T,\ S_{T4} = \left(\frac{2}{15}\right)S_T,\ S_{T5} = \left(\frac{1}{15}\right)S_T$$

$$\frac{(N_{ST1} + (N-1)S_{T2} + (N-2)S_{T3} + (N-3)S_{T4} + (N-4)S_{T5}}{S_T} = N - \frac{4}{3},\ \text{or } P = 1.33$$

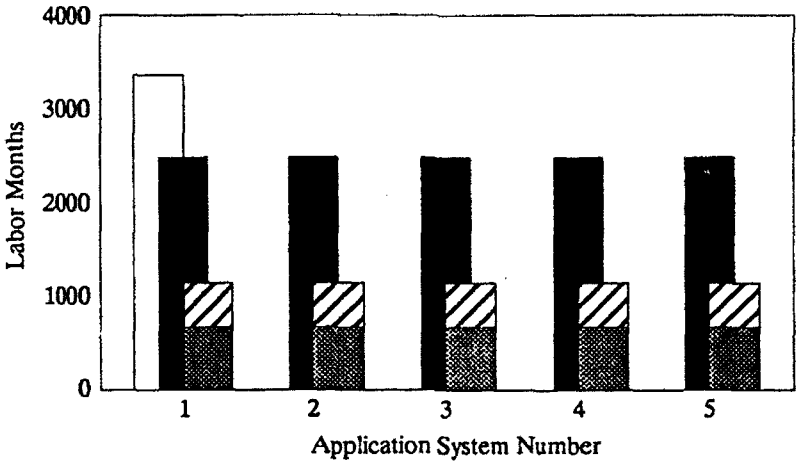
Using these formulas, the cost per application system, for each of a family of five systems, was computed in each of the four cases (regimes). The parametric values used in common for the four regimes are: $S_S = 500$ KSLOC, $S_T = 450$ KSLOC, $C_{VN} = 5.000$ LM/KSLOC, $C_{VR} = 0.5$ LM/KSLOC, $C_{DE} = 7.5$ LM/KSLOC, and $E = 1.0$. All calculations are in LM.

In Table 9, 12,500 LM is the total cost of five application systems without reuse. The cost of money is not included. Figures 8 through 11 illustrate the data in Table 9.

Table 9. Costs for Four Alternative Domain Engineering Investment Regimes

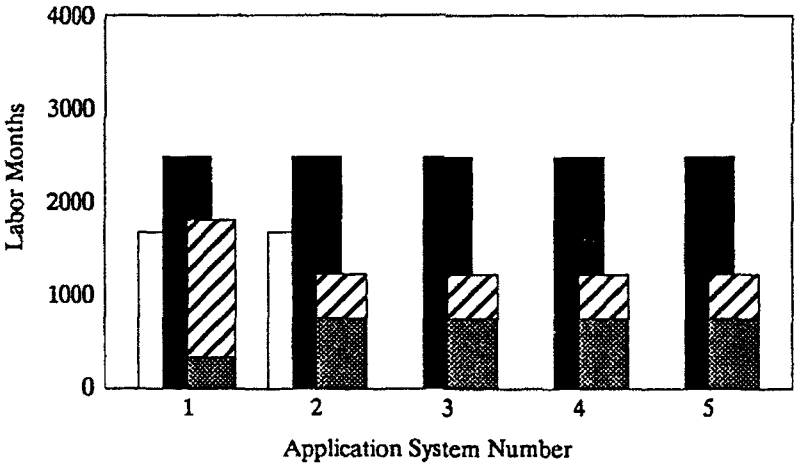
		Case 1		Case 2		Case 3		Case 4	
System	Cost Per System Without Reuse & DE	Domain Engineering Investment (LM)	Cost Per System (LM)	Domain Engineering Investment (LM)	Cost Per System (LM)	Domain Engineering Investment (LM)	Cost Per System (LM)	Domain Engineering Investment (LM)	Cost Per System (LM)9
1	2,500	3,375	1,150	1,687.5	1,825.0	843.75	2,162.5	1,125	2,050
2	2,500	—	1,150	1,687.5	1,234.4	843.75	1,867.2	900	1,735
3	2,500	—	1,150	—	1,234.4	843.75	1,642.2	675	1,555
4	2,500	—	1,150	—	1,234.4	843.75	1,557.8	450	1,510
5	2,500	—	1,150	—	1,234.4	—	1,557.8	225	1,600
Totals(1)	12,500	3,375	5,750	3,375	6,762.6	3,375	8,787.5	3,375	8,450
Savings(2)		6,750 (= 12,500 - 5,750)		5,737.4 (= 12,500 - 6,762.6)		3,718.5 (= 12,500 - 8,787.5)		3,960 (= 12,500 - 8,450)	
Percent Return on Investment = Savings/3,375		200		170		110		120	

It is obvious from this analysis that investing the full cost of domain engineering at the initiation of the domain building effort (case 1) is the least costly course of action with the greatest return on investment. The incremental spending penalty increases as the investment in domain engineering is spread over more and more application systems.



Domain Engineering Investment
Cost Per System With All New Code
Domain Engineering Cost Per System
Application Engineering Cost Per System

Figure 8. Case 1: Domain Engineering Invested All at Once



Domain Engineering Investment
Cost Per System With All New Code
Domain Engineering Cost Per System
Application Engineering Cost Per System

Figure 9. Case 2: Domain Engineering Spread Equally Over Two Increments

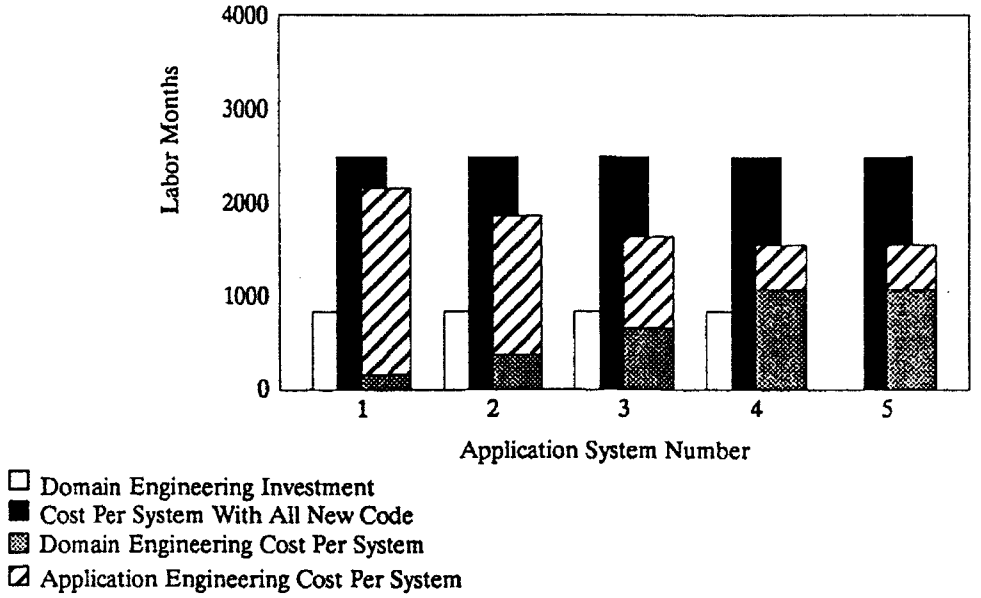


Figure 10. Case 3: Domain Engineering Invested Equally Over Four Increments

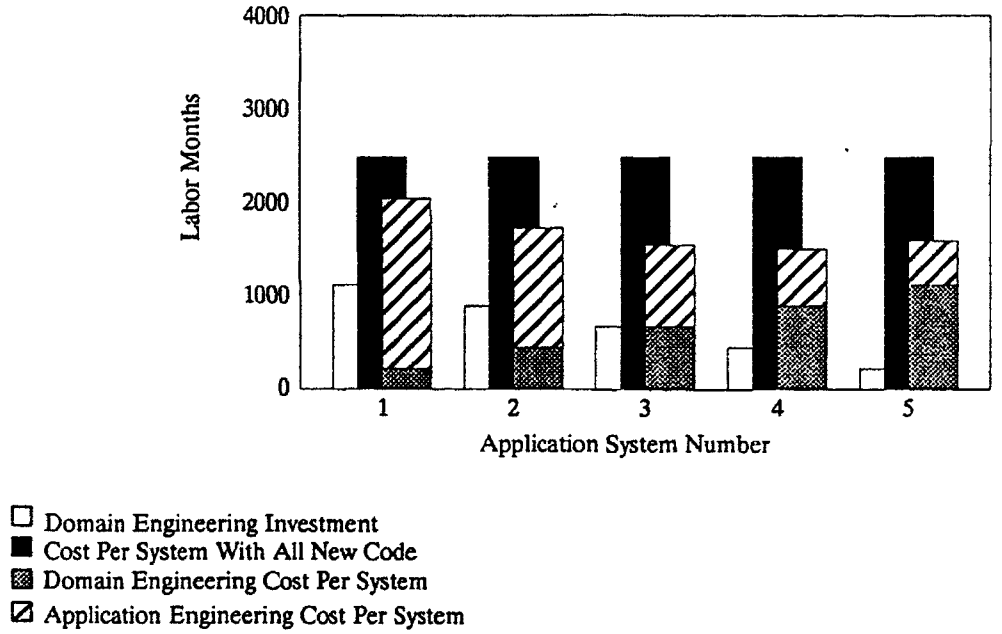


Figure 11. Case 4: Declining Domain Engineering Investment Over Five Increments

3.2.3 The Effects of the Cost of Money

The previous section (3.2.2) on incremental domain engineering did not consider the cost of money (COM). The COM is the interest paid on borrowed funds or the imputed interest, and is an element of cost that many business organizations should consider when making decisions about software development cost.

The calculation of the COM can be organized as an N-by-N array in which the columns correspond to domain engineering investment "streams," and the rows correspond to the costs for each of these streams for each of the N application systems. A stream is an allocated flow of money, COM plus principal, to finance an increment of domain engineering investment. For example, stream 1 (one) begins at application system 1 and corresponds to the domain engineering increment investment made at system 1 and amortized over all N systems. Stream 2 corresponds to the domain engineering increment made at system 2 and is amortized over (N–1) systems, etc. In any cell of the N-by-N computational array, the COM is the product of the portion of investment borrowed for system j under investment stream i and the cost of borrowing for y years at p percent annually.

The formula for the COM in any cell in the N-by-N array (actually only the lower triangular form is used) is:

$$I_{i,j} = \left[a_i \cdot C_T - (j-i) \cdot \left(\frac{a_i \cdot C_T}{N-(i-1)} \right) \right] \cdot [1 + 0.01p)^y - 1] = F_1 \cdot F_2$$

where:

F_1 = Amount of domain engineering investment borrowed for a system in investment stream i.
F_2 = Proportion, COM. For example, 0.36 means that 36 percent of F_1 can comprise COM (i_j), or I_{ij}.

This formula simplifies to:

$$I_{i,j} = a_i \cdot C_T \cdot \left[1 - \frac{j-i}{N-(i-1)} \right] \cdot [(1 + 0.01p)^y - 1]$$

where:

p = Annual percent interest rate.
y = Number of years to which each investment increment is applicable.
C_T = Total domain engineering investment.
a_i = Proportion of $C_T = C_{DE} \bullet S_T$ applied in stream i (The a_i are defined in Section 3.2.2).
T_j = Total COM for application system j, where:

$$T_j = \sum_{i=1}^{N} I_{i,j} \text{ and } i \leq j$$

Two of the four cases, cases 1 and 4, discussed in the previous section are now used as examples of the calculation of the cost of money.

Assume a family of five application systems from the domain in question and that a system can be produced from a domain in four years. Also assume that the current interest rate is eight percent per annum. As previously, all calculations are in LM, and the same parametric values as in the section on incremental domain engineering are used: S_S = 500 KSLOC, S_T = 450 KSLOC, C_{DE} = 7.5 LM/KSLOC, C_{VN} = 5.0 LM/KSLOC, and C_{VR} = 0.5 LM/KSLOC.

The data in Table 9 for case 1 indicates that 3,375 LM is borrowed for four years at eight percent to finance the up-front domain engineering effort as applied to application system 1. Since 675 LM (= 1/5 x 3,375) is amortized by application system 1, then 2,700 LM (= 3,375 - 675) is amortized by system 2 and is borrowed for 4 years (the period of time required for the development of system 2). Similarly, 2,025 LM is borrowed for the next 4 years, and so on. Note that these are the entries in Table 10, which applies to case 1 for stream 1 only. This is because there is only one increment of domain engineering in this case, all up front. Thus, in this case, $S_{T1} = S_T$ and $a_1 = 1$; $a_i = 0$ and $i = 2, 3, 4, 5$.

In case 4, there are five increments of domain engineering as shown in Table 11 and:

$S_{T1} = 1{,}125 = 0.333 \times 3{,}375; \quad a_1 = 0.333$
$S_{T2} = 900 = 0.267 \times 3{,}375; \quad a_2 = 0.267$
$S_{T3} = 675 = 0.200 \times 3{,}375; \quad a_3 = 0.200$
$S_{T4} = 450 = 0.133 \times 3{,}375; \quad a_4 = 0.133$
$S_{T5} = 225 = 0.067 \times 3{,}375; \quad a_5 = 0.067$

Table 10. Cost of Money for Case 1

Domain Engineering Investment Stream											
	Stream 1		Stream 2		Stream 3		Stream 4		Stream 5		
Appl. Sys.	COM	Princi-pal	COM	Princi-pal	COM	Princi-pal	COM	Princi-pal	COM	Princi-pal	Total COM
1	1216.65	675									1,216.65
2	973.32	675									973.32
3	729.99	675									729.99
4	486.66	675									486.66
5	243.33	675									243.33
Total		3,375									3,649.95

Table 11. Cost of Money for Case 4

Domain Engineering Investment Stream											
	Stream 1		Stream 2		Stream 3		Stream 4		Stream 5		
Appl. Sys.	COM	Princi-pal	COM	Princi-pal	COM	Princi-pal	COM	Princi-pal	COM	Princi-pal	Total COM
1	405.55	225									405.55
2	324.44	225	324.44	225							648.88
3	243.33	225	243.33	225	243.33	225					729.99
4	162.22	225	162.22	225	162.22	225	162.22	225			648.88
5	81.11	225	81.11	225	81.11	225	81.11	225	81.11	225	405.55
Total		1,125		900		675		450		225	2,828.85

Table 12 summarizes the COM calculations. The costs have been rounded to the nearest LM.

Table 12. Summary of Cost of Money Cases

		Case No. 1				Case No. 4			
System No.	Cost Per System without Reuse & DE	Domain Engineering Investment (LM)	DE&AE Cost Per System (LM)	Cost of Money (Interest) (LM)	Total (LM)	Domain Engineering Investment (LM)	DE&AE Cost Per System (LM)	Cost of Money (Interest) (LM)	Total (LM)
1	2,500	3,375	1,150	1,217	2,367	1,125	2,050	406	2,456
2	2,500	–	1,150	973	2,123	900	1,735	649	2,384
3	2,500	–	1,150	730	1,880	675	1,555	730	2,285
4	2,500	–	1,150	487	1,637	450	1,510	649	2,159
5	2,500	–	1,150	243	1,393	225	1,600	406	2,006
Totals	12,500	3,375	5,750	3,650	9,400	3,375	8,450	2,480	11,290
Savings		3,100 (= 12,500 – 9,400)				1,210 (= 12,500 – 11,290)			
% Return on Investment = Savings/3,375		92				36			

Figures 12 and 13 illustrate the data in Table 12.

The least costly course of action is to borrow the entire cost of domain engineering at the beginning of the domain building effort (case 1), just as with the previous analysis of incremental domain engineering. The symmetry in the cost of money per system for case 4, with the high amount being for system 3, suggests that a concept similar to the economic lot size of manufacturing may apply.

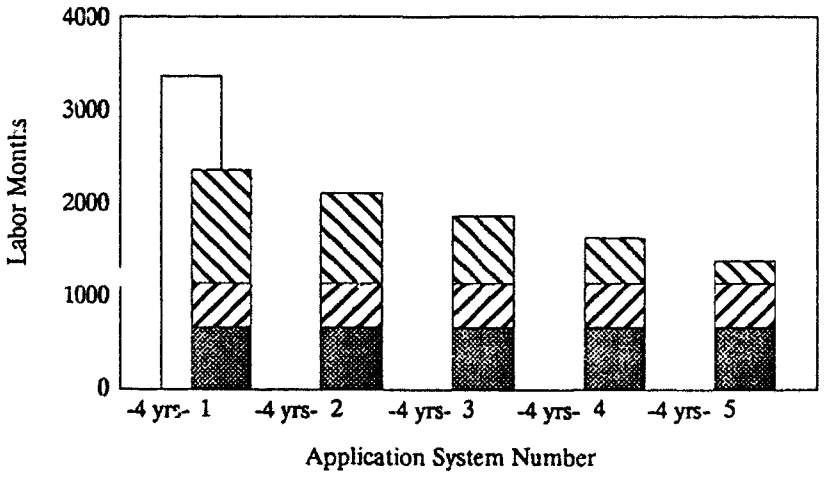


Figure 12. Cost of Money - Domain Engineering is Invested All at Once (Case 1)

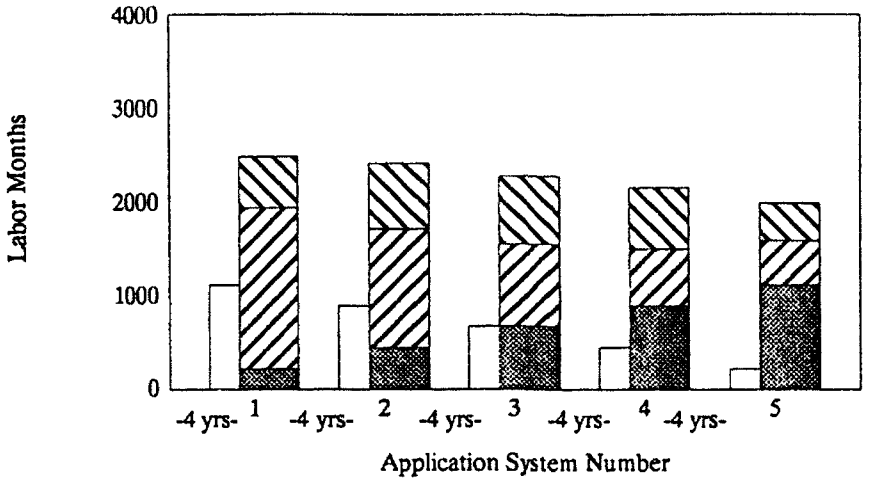


Figure 13. Cost of Money - Declining Domain Engineering Investment Over Five Increments (Case 4)

3.2.4 Alternative Funding Approaches

This section considers the alternatives for funding the creation of software objects that may be used in more than one application system. Software created specifically for one system may be found to useable (albeit with possibly some modifications) in one or more other systems. Alternatively, software may be created especially for use in a family of application systems. Current software engineering practice uses the first case more frequently than the second.

In situations of the first type, formal domain engineering is not done and the costs of creating software objects that are reused in other systems are attributed to the system for which they were created. The cost of reusing these objects in a system other than that for which it was created are added to the costs of making that system. In these cases the costs of creating the software objects are attributed solely to the system for which they were created. This is true even in those situations when part of the new code development effort was devoted to making (at least some of) the code reusable in other systems. The IBM MIS data and the aerospace industry data shown in Figures 2, 3, and 4 is for systems developed according to this mode of operation. In some of the developments represented by the data in those graphs, although some reuse was planned, there was no cost sharing across the developments of several software systems. Rather, the entire cost of creating a software object was attributed to the system in which it was first used.

Situations of the second type, in which at least some software objects are specifically created for use in more than one application, are less frequently found in practice than situations of the first type. In these cases, the costs of developing reusable software objects (objects specifically created for multiple use) are amortized across the cases that use them (or more exactly, over the number of cases in which they are forecasted to be used). This case is analogous to what is done in the development of

a hardware unit; the cost of development is apportioned across the number of copies of that unit that have been forecasted to be sold. The total cost for any one unit is then the sum of the prorated development cost plus the cost of making a copy of the item for each user. That is, the development costs are capitalized across the number of copies of that unit that are expected to be sold. This concept of sharing a capital investment, or equivalently, prorating the cost of a development effort across all units to which it is applicable, is captured in the reuse cost equations presented earlier.

Two current significant examples of the second type of reuse situation, in which reuse is specifically planned for, are now considered. One example is an operating system. One set of source code is instantiated as different systems in the situation in which it is used. This is clearly a situation of multiple use or reuse. Another example is the application software systems employed in the Federal Aviation Agency's (FAA) enroute control centers. This software processes radar data, produces tracks which are observed by the air traffic controller, and performs other functions for the air traffic control system. The same source code is used as the basis for the software operating in each of the control centers. This source code gives rise to the code operating in each center's computers; it is unique to that center. The uniqueness is with respect to such items as the configuration of the airways under the jurisdiction of that center and the configuration of the radars and other equipment that it uses. The code operating in each center is updated every two months to reflect the changes, if any, made with respect to these items.

The reuse unit cost equation and its variants are now considered with respect to how they can model various alternative approaches for funding the creation of software objects designed for use in one more than one application software system. The case of software developed for the government is focused on because of the different ways in which the capital costs of domain engineering can be viewed. Reuse can occur among variants of the same system or for the same or similar functions in different systems. An example of the first case is the reuse of objects in the avionics software of various versions of the F–22 (also known as the advanced tactical fighter) aircraft (i.e., an F–22A, F–22B ,etc.). An example of the second is the use of certain software objects in different aircraft, such as the F–22 and the LH (helicopter). Recently, various government initiatives, which included the participation of industry representatives, considered such possibilities.They include the Joint Integrated Avionics Working Group (JIAWG) in the late 1980's and 1990, and the Joint Logistics Commanders' (JLC) software reusability panel in early 1991.

Typically, at least until recently, each system built for the government has been viewed as unique. Every software object built in such a system is paid for under the contract for that system, even if it may be reused in another system. Therefore, if the software is reused in another system, that system's developer obtains it free of charge for incorporation into his system. Of course, he does have to charge off the costs of incorporating such objects into his system. A major disadvantage of this (traditional) approach is that there is no incentive to spend the extra increment of development cost required to make a software object more reusable, i.e., to design it specifically for multiple use. In general, there is little incentive for the contractor to consider the likely variants to his system that would facilitate his creation of software objects that could be reused in other systems. Both the JIAWG and the JLC panel, cited above, have considered various ways to create incentives for both the creation of reusable software objects and their reuse in systems built for the government. These approaches are not considered further.

Usually, a government program manager has had little incentive to have the software for his system created so that it is reusable. This situation may change. One recent innovation in the DoD's development management structure that may be helpful is the creation of the position of program executive officer (PEO) function. Such a person has the financial responsibility for several programs and hence

has a strong incentive for minimizing the costs over all of them. He could employ a number of different strategies to do so. One is to pay for the domain engineering effort that is applicable to several projects that are expected to be developed approximately in parallel, but before they occur. Another might be to pay for the domain engineering capital investment that would be applicable to the portion of each of several versions of the same system that are anticipated to be common among them. The reuse unit cost equation provided in this report, including the domain engineering term, is applicable to the point of view of the PEO. If the contractor received the results of the domain engineering effort free of charge, then he would not have to consider the costs of domain engineering. In other situations, he is advised to do so, however. For example, he might find it useful to invest in maximizing the degree of reusability of some of the software he created for a given version of a system in order to increase his competitive advantage for bidding on future systems.

The reuse unit cost equation provided in this report can be used to consider reuse, or potential reuse, from different points of view. It provides a framework for considering the economics of reuse from the point of view of someone who wants to minimize the development costs of a family of software systems. The framework can also be used by someone concerned with a more "local" cost minimization, applicable to one or several, but not necessarily all, of a family of application software systems.

This page intentionally left blank.

4. MODELING REUSE OF REQUIREMENTS AND DESIGN IN ADDITION TO CODE

This section explicitly considers the reuse (or multiple use as defined earlier) of RSOs in addition to code. The basic reuse economics model, and its variant which covers incremental domain engineering, deal with code reuse. Recall that the factor R stands for the proportion of code reuse in an application system of size S_S. The reuse of a unit of code includes the reuse of the corresponding software objects from which it was derived, the requirements, and the design. This section addresses cases that reuse requirements and/or designs, but not necessarily code.

The models presented in this section can be used as an aid in evaluating some aspects of the economics of software reengineering. For example, a software system coded in FORTRAN might be reengineered so that it can be reimplemented in Ada. In this case, the requirements and at least some of the design might be reused, but the code would not be. The models presented here can be used to estimate the impact of the reuse of requirements and design in this situation.

4.1 DERIVATION OF SIZE RELATIONSHIPS

New code can be created from new requirements and new design, from reused requirements and new design, or from reused requirements and reused design. However, reused code can not be derived from either new requirements or new design. This statement of simple logic underlies the development of the mathematical relations provided in this section.

The amount of reuse during later phases of development is upper-bounded by the amount of reuse during earlier phases. For example, the amount of design reuse cannot exceed the amount of reuse of requirements (when both quantities are expressed in terms of equivalent SLOC or in terms of their respective proportions of total SLOC in the application software system). Suppose S_{RR} is the source statement equivalent of reused requirements, S_{RD} is the source statement equivalent of reused design, S_R is the source statement count of reused code, and S_S is the size of the application system overall.

Then:

$$S_{NR} + S_{RR} = S_S$$

$$S_{ND} + S_{RD} = S_S$$

$$S_N + S_R = S_S$$

Since reused code cannot be derived from either new requirements or new design, the following relationships are true:

$$S_{RR} \geq S_{RD} \geq S_R$$

$$S_N \geq S_{ND} \geq S_{NR}$$

$$R_{RR} \geq R_{RD} \geq R$$

where $R_{RR} = S_{RR}/S_S$, the proportion of requirements reuse, $R_{RD} = S_{RD}/S_S$, the proportion of design reuse, and $R = S_R/S_S$, the proportion of code (and of testing) reuse.

Figure 14 graphically shows the relationships among new and reused objects in terms of size.

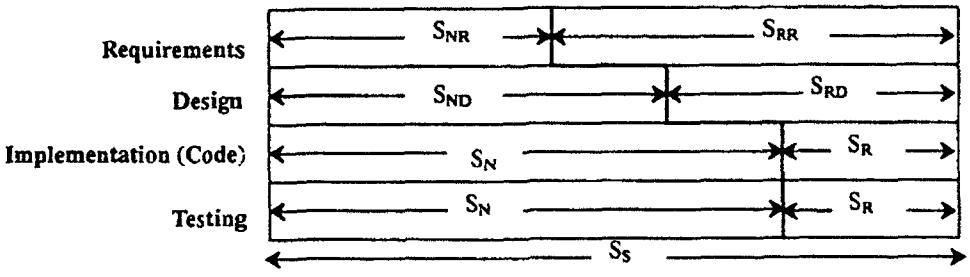


Figure 14. New and Reused Objects at Different Levels

4.2 APPLICATION OF COST RELATIONSHIPS

This section develops the cost relationships for the various types of new and reused software objects.

The symbols for the unit costs (in LM/KSLOC, where the KSLOC represent the equivalent source statements for the reused objects) for the various reused objects are shown in Table 13.

Table 13. Unit Costs of New and Reused Objects

Phase/Activity	Unit Costs New Objects (N)	Unit Costs Reused Objects (R)
Requirements (R)	C_{VNR}	C_{VRR}
Design (D)	C_{VND}	C_{VRD}
Implementation (Code) (I)	C_{VNI}	C_{VRI}
Testing (T)	C_{VNT}	C_{VRT}

Suppose that the unit cost of new code, C_{VN}, is 3.5 LM/KSLOC (286 SLOC/LM) and that there is an equal reuse of requirements, design, code, and testing. Let the breakdown of development effort be 20 percent for requirements, 30 percent for design, 20 percent for implementation (coding), and 30 percent for testing, so that the unit cost equation for new code is expressed numerically as:

$$3.5 = 0.7 + 1.05 + 0.7 + 1.05 \quad \text{LM/KSLOC}$$

which corresponds to:

$$C_{VN} = C_{VNR} + C_{VND} + C_{VNI} + C_{VNT}$$

This value of C_{VN} is the base value of C_{VN}. Now suppose that there is an unequal reuse of requirements, design, and code. Let R = 0.5, R_{RR} = 0.7, and R_{RD} = 0 .7. Then:

$$C_{VRR} = (1 - R_{RR})C_{VNR} = (.3)(.7) = 0.210$$

$$C_{VRD} = (1 - R_{RD})C_{VND} = (.3)(1.05) = 0.315$$

Then using the same equation for C_{VN} developed above:

$$C_{VN} = 1.75 + 1.05\frac{1-0.7}{1-0.5} + 0.315\frac{0.7-0.5}{1-0.5} + 0.7\frac{1-0.7}{1-0.5} + 0.210\frac{0.7-0.5}{1-0.5}$$

$$C_{VN} = 1.75 + 0.03 + 0.126 + 0.42 + 0.084 = 3.01 \text{ LM/KSLOC}$$

which is equivalent to 322 SLOC/LM. Thus, some requirements and design reuse causes the new code productivity to increase from 286 to 322 SLOC/LM.

Overall, the total cost of application engineering is:

$$C_A = C_{VN}S_N + C_{VR}S_R = C_N + C_R$$

New code can be derived from reused requirements or reused design, but reused code cannot be derived from new requirements or new design. Therefore the total cost of reused code is:

$$C_R = C_{VR}S_R = (C_{VRR} + C_{VRD} + C_{VRI} + C_{VRT})S_R$$

where C_{VR} is the unit cost of reusing requirements. The total cost of new code is (see Figure 14):

$$C_N = C_{VN}S_N = C_{VNR}S_{NR} + C_{VRR}(S_{RR} - S_R) + C_{VND}S_{ND} + C_{VRD}(S_{RD} - S_R) + (C_{VNI} + C_{VNT})S_N$$

The following relationships hold (from the above equations):

$$S_N = (1 - R)S_S; \quad S_{ND} = (1 - R_{RD})S_S; \quad S_{RD} - S_R = (R_{RD} - R)S_S;$$

$$S_{RR} - S_R = (R_{RR} - R)S_S; \quad S_{NR} = (1 - R_{RR})S_S$$

Substituting into the previous equation for C_N and dividing through by (1-R):

$$C_{VN} = (C_{VNI} + C_{VNT}) + C_{VND}\frac{1-R_{RD}}{1-R} + C_{VRD}\frac{R_{RD}-R}{1-R} + C_{VNR}\frac{1-R_{RR}}{1-R} + C_{VRR}\frac{R_{RR}-R}{1-R}$$

This is the general cost relation equation for new code under the condition of different proportions of various reused objects. Note that if $R_{RR} = R_{RD} = R$, that is, if all of the new code is derived from (corresponds to) new requirements and design, then $C_{VN} = C_{VNI} + C_{VNT} + C_{VND} + C_{VNR}$, as would be expected. This is, the general cost relation for new code reduces to the cost for new code when all of the new code is derived from new requirements and new design.

4.3 GENERALIZATION OF LIBRARY EFFICIENCY METRIC

This section generalizes the concept of library efficiency to cover the case in which objects other than code are reused but the code may not be. The proportion of code reuse can be less than the proportion

of requirements and/or design reuse. R_E is the effective (overall) reuse proportion. It is a weighted sum of the reuse proportions of requirements (R_{RR}), design (R_{RD}), and code (R). Therefore, the effective (overall) reuse proportion:

$$R_E = R_{RR} \cdot \frac{C_{VRR}}{C_{VR}} + R_{RD} \cdot \frac{C_{VRD}}{C_{VR}} + R \cdot \frac{(C_{VRI} + C_{VRT})}{C_{VR}}$$

Let $C_{VRI} + C_{VRT} = C_{VRIT}$. Then the equation for R_E can be written as:

$$R_E = R_{RR} \cdot \frac{C_{VRR}}{C_{VR}} + R_{RD} \cdot \frac{C_{VRD}}{C_{VR}} + R \cdot \frac{C_{VRIT}}{C_{VR}}$$

Further note that:

$$\frac{C_{VRR}}{C_{VR}} + \frac{C_{VRD}}{C_{VR}} + \frac{C_{VRIT}}{C_{VR}} = 1$$

Thus, when $R_{RR} = R_{RD} = R$, the case of full (code) reuse (including the reuse of requirements and design), then $R_E = R$, as it should. R_E is a generalization of the proportion of reuse, R, and is used in the generalization of the basic unit cost equation as shown in the following subsections.

Then $S_{RE} = R_E \cdot S_S$. If $R_{RR} = R_{RD} = R$, then $R_E = R$ and $S_{RE} = S_R$.

Now let $K = S_T/S_S$ be as originally defined, the relative library capacity.

Therefore, the general expression for library efficiency that covers the case of reuse of objects other than code as well as the reuse of code is:

$$E = \frac{R_E}{K} = \frac{\frac{S_{RE}}{S_S}}{\frac{S_T}{S_S}} = \frac{S_{RE}}{S_T} = \frac{R_E \cdot S_S}{S_T}$$

This definition of library efficiency represents a generalization of the original definition that takes into account the reuse of objects when code is not necessarily reused. If $R_{RR} = R_{RD} = R$, then $R_E = R$, and $E = S_R/S_T$, as originally defined in Section 2.2.2.

4.4 GENERALIZATION OF N

The factor N was defined earlier as the number of application systems in the family. It was used as the number of systems over which an up-front investment in domain engineering is amortized. It presumed code reuse and reuse of the corresponding requirements and design. This section generalizes N to N_E, the number of equivalent application systems for amortization purposes when the amount of code reused may be less then the amount of design or requirements (as considered in the previous section).

The unit cost of domain engineering:

$$C_{DE} = C_{DER} + C_{DED} + C_{DEIT}$$

where C_{DER} is the unit cost for creating reusable requirements, C_{DED} is the unit cost for creating reusable design, and C_{DEIT} is the unit cost for creating reusing implementation (code) and test. The prorated unit cost is:

$$C_{DE} * N_E = C_{DER} * N_R + C_{DED} * N_D + C_{DEIT} * N$$

Therefore:

$$N_E = \frac{C_{DER}}{C_{DE}} * N_R + \frac{C_{DED}}{C_{DE}} * N_D + \frac{C_{DEIT}}{C_{DE}} * N$$

where N_E is the number of equivalent (to full) application systems considering the reuse of requirements and design objects as well as code objects, N_R is the number of application systems over which the reused requirements are prorated, N_D is the number of systems over which the unit cost of the reused design is amortized, and N is the number of systems over which the unit cost of implementation and testing is amortized. N_E can be viewed as the weighted sum of the number of each type of RSOs used in the new application system. It is also true that:

$$1 \leq N \leq N_D \leq N_R$$

If $N_R = N_D = N$, then, $N_E = N$, as it should. The generalization of N and R leads to a generalization of the basic unit cost equation as shown in the next subsection.

4.5 GENERALIZATION OF BASIC UNIT COST EQUATION

The basic unit cost equation with up-front domain engineering is being generalized to take into account the reusable requirements and/or design without necessarily having corresponding code reuse. The approach is to substitute the factors R_E and N_E for R and N, respectively. Then:

$$C_{US} = \frac{C_{DE}}{N_E} * K + C_{VN} - (C_{VN} - C_{VR})R_E$$

where C_{VN} is defined in its generalized form:

$$C_{VN} = (C_{VNI} + C_{VNT}) + C_{VND}\frac{1-R_{RD}}{1-R} + C_{VRD}\frac{R_{RD}-R}{1-R} + C_{VNR}\frac{1-R_{RR}}{1-R} + C_{VRR}\frac{R_{RR}-R}{1-R}$$

This page intentionally left blank.

5. THE EFFECT OF REUSE ON QUALITY

Reuse enhances the quality of an application system principally because of the increased opportunity it provides for error discovery. Each time reusable code is used in a new application software system, it passes through the integration and system test process again. Thus, an additional opportunity is provided for error discovery and removal. This section focuses on the reuse of code and, implicitly, the reuse of the requirements and design from which it came.

This section presents a mathematical model that can be used to predict the quality enhancement expected as a function of R, the proportion of code reuse. The model relates the number of errors in a software product at time of delivery (i.e., the latent error content) to R. A software development process consists of a set of activities in which errors may be discovered. The difference between the quality of new and reused code is principally due to the fact that the reused code undergoes integration testing N times for use in N application systems, while the new code (for an application system) undergoes testing just once. Both the new and reused code components of an application system are presumed to go through the other error discovery activities the same number of times.

Let D_{VR} be the latent error content of some code when placed in a software reuse library for initial reuse or when picked up from the software system for which it was developed for reuse in a new application. Let D_{VN} be the latent error content of a software product composed entirely of new code. Let both D_{VN} and D_{VR} be measured in errors per KSLOC. These parameters represent the latent error content of their categories of software, i.e., the error content of the software when it is delivered to the customer.

It is assumed that the code to be reused in an application software system has gone through the complete development process before this reuse (whether it is provided from a library or is taken from a prior system). The code to be reused in an application system is presumed to go through integration and system test in the development of a (new) application system. However, it is assumed the code does not go through the earlier error discovery steps in design and code inspections and unit test earlier in the development process as the new code component of the new application system is expected to.

We have the following expression for D_{Ri}, the latent error density in the new application system which includes the *i*th use of the "reused" code:

$$D_{Ri} = D_{VN} * (1 - R) + D_{VR} * R * p^i$$

where R is the proportion of code reused (on the average over the N planned uses of the reused code). Let:

$$p = \frac{\text{Latent error content}}{\text{Errors discovered and removed during the integration and system test process + latent error content}}$$

where the development process is preliminary (top level) and detailed design (including internal reviews and preliminary and critical design reviews), implementation (including code inspections and error correction), CSC integration test, and CSCI (system) test.

In the case of the first use after the creation of the reusable software, $p^1 = p$, and:

$$D_{Ri} = D_{R1} = D_{VN} * (1 - R) + D_{VR} * R * p$$

In the case of the second use:

$$D_{Ri} = D_{R2} = D_{VN} * (1 - R) + D_{VR} * R * p^2$$

An example value of p can be derived from data in (Gaffney 1984). It is presented in Table 14.

Table 14. Example Values of Error Discovery Percentages

Phase/Activity	Percent of Lifetime Errors
High-Level (Preliminary) Design Inspections	7.69
Detailed Design Inspections	19.70
Code Inspections	23.93
Unit Test	20.88
Integration Test	14.27
System Test	7.92
Latent error content	5.61
Total	100.00

In this case:

$$p = \frac{5.61}{14.27 + 7.92 + 5.61} = 0.2018$$

The thinking behind the factor $p \bullet D_{VR}$ is as follows. After the reusable code had been developed for the library or for use in some prior application system (from which it is taken for inclusion in the library), it still had some latent errors (such as 5.61 % of the errors) that had been injected during the development process (as in the example case summarized in Table 14). Upon the first of N uses, the code to be reused goes through integration and system tests, thus removing a proportion given by:

$$\frac{14.27 + 7.92}{14.27 + 7.92 + 5.61} = 0.7982$$

and leaving a proportion of $1.0 - 0.7982 = 0.2018 = p$ times the latent error content of the code after it had been developed and put into the library. The same relative percentage of error reduction occurs when going from the first use to the second use, and so on.

Let L be the latent error content of a software product that is composed in part of reused components, relative to one composed of entirely new code. Thus, for a product having no reused components 1. In general, $0 \leq L \leq 1$ (under the practical assumption that reuse does not add to the laten. ror

content). The quantity L may be seen as equal to the proportion of new code times the latent errors relative to the all new code case, plus the proportion of reused code times the latent errors relative to the all new code case. Thus for the *i*th instance of use out of N:

$$L_i = \frac{D_{VN}}{D_{VN}} \cdot (1-R) + \frac{D_{VR}}{D_{VN}} \cdot R \cdot p^i = (1-R) + \frac{D_{VR}}{D_{VN}} \cdot R \cdot p^i$$

Now, if we assume that the error discovery profile shown in Table 14 applies to both the new code created for an application system and the code to be reused in the application, then $D_{VR} = p \cdot D_{VN}$. In this case, the equation for L_i can be written as:

$$L_i = (1-R) + R \cdot p^{i+1}$$

The factor L is the average quality enhancement (latent error reduction) over the N instances. Thus:

$$L = \left(\sum_{i=1}^{N} L_i \right) / N$$

And:

$$L = (1-R) \cdot \frac{N}{N} + \frac{R}{N} \left(\sum_{i=1}^{N} p^{i+1} \right)$$

This expression for L can be simplified by recognizing the similarity of the factor Σp^{i+1} to a geometric series. Indeed:

$$\sum_{i=1}^{N} p^{i+1} = p^2 + p^3 + \cdots + p^{N+1}$$

A similar geometric series with $a = 1$ as the first term and with a common ratio of p is:

$$1 + p + p^2 + \cdots + p^{N-1} = \frac{1-p^N}{1-p}$$

Therefore:

$$\sum_{i=1}^{N} p^{i+1} = p^2 \cdot \left(\frac{1-p^N}{1-p} \right)$$

Consequently, we may write:

$$L = (1-R) + p^2 \cdot \frac{R}{N} \cdot \left(\frac{1-p^N}{1-p} \right)$$

As N gets larger, L tends to (1-R).

To gain an appreciation for the speed of convergence of L to its limit (1-R), consider the following example in Table 15 in which p = 0.20.

Table 15. Sample Values of L for p = 0.20

L	N
1-0.96R	1
1-0.975R	2
1-0.983R	3
1-0.988R	4
1-0.9900R	5
1-0.9950R	10

L tends asymptotically to (1-R) fairly quickly as N grows. Figure 15 presents plots of L as a function of N for p = 0.20. L approaches the (1-R) exponentially. As p gets larger (less effective error discovery and more error-prone code), the value of N at which L is close to (1-R) becomes larger. This would appear to be commensurate with engineering intuition. The model holds under the important assumption that the causes of the errors detected during the various discovery stages are removed more or less contemporaneously with their discovery.

An analyst, software engineer, or manager can use the formula for L, or its approximation given in Table 15, to estimate the impact of extensive reuse in an application system as described. First, the latent error content of new software is estimated, based on past experience with similar kinds of code or using an approach like the one implemented in the software early error prediction (SWEEP) model (Gaffney and Pietrolewicz 1990). Then, this figure is reduced by the factor L and computed as described above.

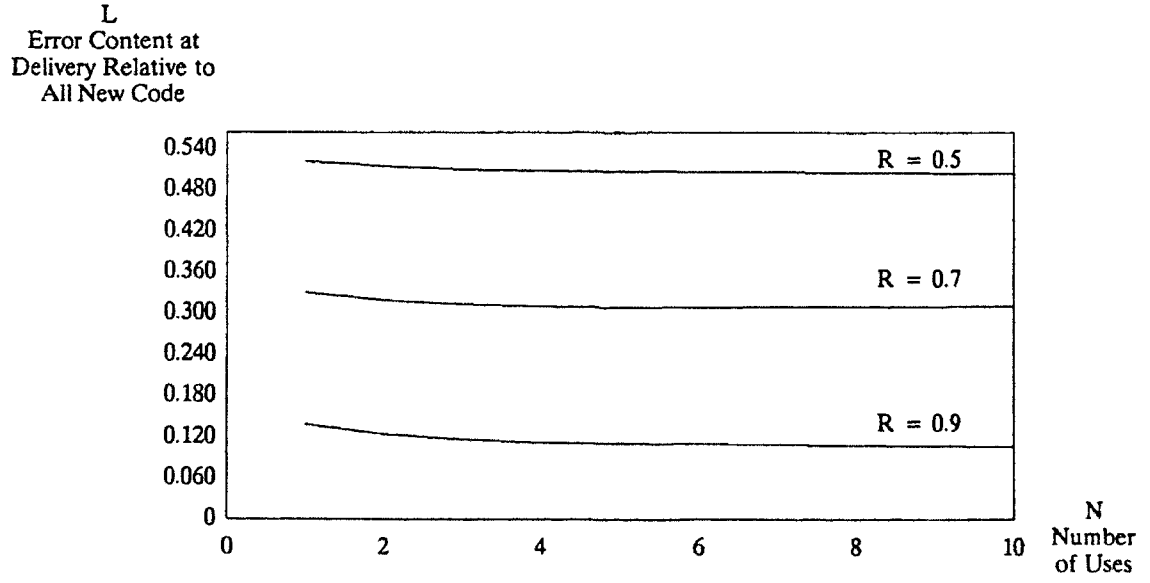


Figure 15. Average Relative Error Content Versus Number of Uses (p = 0.20)

6. SUMMARY

Ad hoc reuse offers savings in development costs, but more substantial savings can be achieved when a systematic reuse process is used. The very nature of the systematic reuse process requires software developers to consider not only the current version of the system being developed but future versions as well. This consideration has a large impact on the supportability of the system, with its attendant cost consequences.

The incremental funding of the domain engineering investment generally results in lower returns on investment than up-front funding but has the advantage of conserving capital until required. It recognizes that it may not be possible to fully describe a domain before any of the systems in the domain family have been constructed.

Some of the models presented in this report can be used to gain insight into the aspects of the economics of software reengineering. For example, the models could be used, where appropriate, to estimate the cost of a new application system that would be created by redesigning (and recoding) an existent system. In creating such a system, the developer reuses requirements and design but not code.

The economics models presented in this report not only demonstrate the economics impact of systematic reuse, but also provide as a means to learn about applying a systematic reuse process. With these models, the user can explore the costs and benefits of an investment in a domain. "What if" explorations can help support business decisions.

The principle models developed in this report are summarized below. The basic unit cost equation is:

$$C_{US} = \frac{C_{DE}}{N} \cdot K + C_{VN} - (C_{VN} - C_{VR}) \cdot R$$

where the library efficiency is given by:

$$E = \frac{R}{K} = \frac{S_{R/S_S}}{S_T/S_S} = \frac{S_R}{S_T}$$

When the reuse of software objects in addition to code, i.e., requirements and design, is specifically considered, the basic unit cost equation generalizes to:

$$C_{US} = \frac{C_{DE}}{N_E} \cdot K + C_{VN} - (C_{VN} - C_{VR})R_E$$

where the generalized reuse proportion is:

$$R_E = R_{RR} \cdot \frac{C_{VRR}}{C_{VR}} + R_{RD} \cdot \frac{C_{VRD}}{C_{VR}} + R \cdot \frac{C_{VRIT}}{C_{VR}}$$

and the generalized number of application systems is defined by:

$$N_E = \frac{C_{DER}}{C_{DE}} \cdot N_R + \frac{C_{DED}}{C_{DE}} \cdot N_D + \frac{C_{DEIT}}{C_{DE}} \cdot N$$

and the library efficiency generalizes to:

$$E = \frac{R_E}{K} = \frac{\frac{S_{RE}}{S_S}}{\frac{S_T}{S_S}} = \frac{S_{RE}}{S_T} = \frac{R_E \cdot S_S}{S_T}$$

The return on investment is:

$$ROI = \left[\frac{N \cdot E \cdot (C_{VN} - C_{VR})}{C_{DE}} - 1 \right] \cdot 100 = \left[\frac{N}{N_0} - 1 \right] \cdot 100$$

and the break-even number of systems is:

$$N_0 = \frac{C_{DE}}{(C_{VN} - C_{VR})E}$$

which, when the reuse of objects other than code (RSOs) is considered, generalizes to:

$$N_0 = \frac{C_{DE}}{C_{VN} - C_{VR}} + P$$

where E is assumed to be equal to 1.0 and where P is the additional number of application systems required to break even due the use of incremental domain engineering. Finally, the average quality enhancement because of software reuse (the latent error reduction over N application systems) is given by:

$$L = (1 - R) + p^2 \cdot \frac{R}{N} \cdot \left(\frac{1 - p^N}{1 - p} \right)$$

where p is the proportional latent error reduction in the reused code from one reuse application to the next in the series of N application systems from the domain.

The equations presented here summarize the reuse economics models described in this report. The Consortium will implement a spreadsheet reuse economics modeling capability in 1991. Users will be able to evaluate a variety of reuse situations.

REFERENCES

Albrecht, A.J. 1989 — Personal communication.

Albrecht, A.J., and J.E. Gaffney, Jr. 1983 — Software Function, Source Lines of Code, Development Effort Prediction: A Software Science Validation. *IEEE Transactions on Software Engineering SE-9.*

Campbell, G.H. 1990 — *Synthesis Reference Model* Version 01.00.01. SYNTHESIS_REF_MODEL-90047-N. Herndon, Virginia: Software Productivity Consortium.

Campbell, G.H., S.R. Faulk, and D.M. Weiss 1990 — *Introduction to Synthesis.* Version 01.00.01 INTRO_SYNTHESIS_PROCESS-90019-N. Herndon, Virginia: Software Productivity Consortium.

Cruickshank, R.D., and J.E. Gaffney Jr. 1990 — *Synthesis Economic Analysis And Reuse Economic Model Improvement* Version 01.00.01. SYNTHESIS_ECON_MODEL-90020-MC. Herndon, Virginia: Software Productivity Consortium.

Cruickshank, R.D., and M. Lesser 1982 — An Approach to Estimating and Controlling Software Development Costs. *The Economics of Data Processing.* New York, New York: Wiley.

Gaffney, J.E. Jr. 1989 — *An Economics Foundation for Software Reuse* Version 1.0, SW-REUSE-ECONOM-89040-N. Herndon, Virginia: Software Productivity Consortium.

1984 — On Predicting Software Related Performance of Large-Scale Systems. *International Conference of the Computer Measurement Group.* CMG XV. San Francisco, CA.

1983 — Approaches to Estimating and Controlling Software Costs. *1983 International Conference of the Computer Measurement Group,* CMG XIV. Washington, D.C.

Gaffney, J.E. Jr., and R.D. Cruickshank 1991a — *Code Counting Rules and Category Definitions/Relationships* Version 02.00.04, CODE_COUNT_RULES-90010-N. Herndon, Virginia: Software Productivity Consortium.

Gaffney, J.E. Jr., and R.D. Cruickshank 1991b — The Measurement of Software Product Size: New and Reused Code. *Third Annual Oregon Workshop on Software Metrics*, Silver Falls, Oregon. To be published in a forthcoming issue of *Software Engineering: Tools, Techniques, and Practice.*

Gaffney, J.E. Jr., and T. Durek 1988 — *Software Reuse -Key to Enhanced Productivity: Some Quantitative Models* Version 1.0, SPC-TR-88-015. Herndon, Virginia: Software Productivity Consortium.

1991 — Software Reuse—Key to Enhanced Productivity: Some Quantitative Models. *The Economics of Information Systems and Software.* pp 204-219. R. Veryard, ed. Butterworth-Heinemann, Oxford, England.

Gaffney, J.E. Jr., and J. Pietrolewicz 1990 — An Automated Model for Software Early Error Prediction (SWEEP). *Thirteenth Minnowbrook Workshop on Software Engineering,* Blue Mountain Lake, New York.

Parnas, D. 1976 — Design of Program Families. *IEEE Transactions on Software Engineering.* 2, 1:1.

Software Productivity Consortium 1991 — *Software Measurement Guidebook,* SPC-91060-MC, Version 01.00.04. Herndon, Virginia.